1985

A High Level Programming Language and Graphics Simulator for the Heathkit ET-18 Hero Robot

Kenneth J. Sizemore
University of Central Florida

A HIGH LEVEL PROGRAMMING LANGUAGE AND GRAPHICS SIMULATOR FOR THE HEATHKIT ET-18 HERO ROBOT

BY

KENNETH JAMES SIZEMORE
B.S.E., University of South Florida, 1979

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science
in the Graduate Studies Program
of the College of Engineering
University of Central Florida
Orlando, Florida

Summer Term
1985

ABSTRACT

As modern industrial robots are used more in industry, the need for qualified programmers and operators will increase. Inexpensive robots are available for use as training systems, but these robots typically lack the high level tools, such as high level languages and graphics simulators, that are available for the industrial robots.

A high level programming language and a graphics simulator were developed for one of these training robots, the Heath ET-18 Hero Robot. The programs were designed to be executed on an IBM Personal Computer. The software was developed as part of an overall system which also includes a compiler and a program for downloading software to the Hero over a serial interface.

ACKNOWLEDGEMENTS

I would like to express my appreciation to the following people for their help and support in the development of this thesis: Dr. Christian S. Bauer, Dr. John Biegel, Brian Cardinal, Vick DeGiorgio, Mike Groberg, Robert Gundal, Dr. Harold Klee, Clint Lyttle, Norman Marler, Bob Pettigrew, Joe Rubel, and Robert Thornhill.

I thank the Martin Marietta Corporation for allowing me to participate in the Industrial Associates Graduate Work/Study Program to pursue my master's degree in Computer Engineering at the University of Central Florida.

I especially thank Dan Fischoff, the codeveloper of the language and programming system discussed in this thesis. His efforts, spirit, and teamwork helped make the total system greater than the separate parts.

Finally, my greatest thanks goes to my wife, Darla. Without her continuous support, encouragement, and understanding, this project could not have been accomplished.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

INTRODUCTION

Modern industrial robots perform well in repetitive and dangerous jobs and can greatly improve the productivity of manufacturing industries when properly applied. The main advantage of robots is their flexibility: they can cope with multiple products on one manufacturing line or cell, and can be reprogrammed to reflect product modifications. As more and more robots are brought into the factory, the need for trained robot operators and programmers will increase dramatically. Effective methods must be developed to train operators in this field.

Many small educational robot training systems are available in the marketplace, such as the Heathkit Hero Robot and the Rhino XR-2 Robot (Washburn 1984). Unfortunately, the programming systems for these training robots are more complex and less powerful than the programming systems available for most industrial robots. Most industrial robots can be equipped with high level languages and programming systems to improve the productivity of robot operators.

In order to make the Hero Robot a more effective educational tool, a high level programming system was developed. The programming system, which operates on an IBM Personal Computer, consists of a high level language, a

compiler, a graphics simulator, and a program to download instructions from the personal computer to the Hero Robot via a serial interface.

The programming system is called HICLASS, an acronym for the Hero Instruction Compiler, Language, And Simulation System. This system was developed jointly with Mr. Daniel Fischoff, a graduate student in the Industrial Engineering Department at the University of Central Florida. The Hero High Level Language was a combined effort of the author and Fischoff, who is the coauthor of the sections titled "Hero Programming Language" and "Hero High Level Language Reference Manual" of this thesis. The remainder of the project was broken into two distinct paths. The development of the graphics simulator was the responsibility of the author. Fischoff developed the compiler and loader for the high level language, which are described in detail in his thesis titled A High Level Language and Compiler for the Heathkit ET-18 Hero Robot. The graphics simulator is discussed in "Graphics Simulator Design" and "Development of the Graphics Simulator Software," followed by the user manual for the graphics simulator. Conclusions and recommendations are discussed in the final two sections.

Description of the Hero Robot

The Heathkit ET-18 Hero Robot is one of the educational systems available to train robot operators and programmers.

The Hero is capable of motion in five axes and has sensors for detecting sound, light, motion, and distance. In addition, the robot has a synthesized phoneme-based speech system.

The Hero is a mobile robot which can be driven forward and backward. The steering wheel can be turned 90 degrees to the right and 90 degrees to the left of the straight ahead position. The Hero has a head which rotates 350 degrees. Attached to the back of the head is an arm which can be raised and lowered 150 degrees in the vertical plane by a shoulder motor. An extender motor extends and retracts the arm 5 inches. A wrist pivot motor moves the gripper 90 degrees above or below the axis of the arm and a wrist rotate motor moves the gripper through 350 degrees. The gripper motor opens the gripper 3.5 inches. An illustration of the Hero Robot is shown in Figure 1 (Heath Company 1982).

Overview of the Hero High Level Language

The Hero High Level Language was developed to provide an English-based system for the robot. This type of language is easier to use and learn than the standard machine code system of the robot. In addition to a simpler command format, the language allows the user to express movements in terms of angles and distances as opposed to an internal hexadecimal position count.

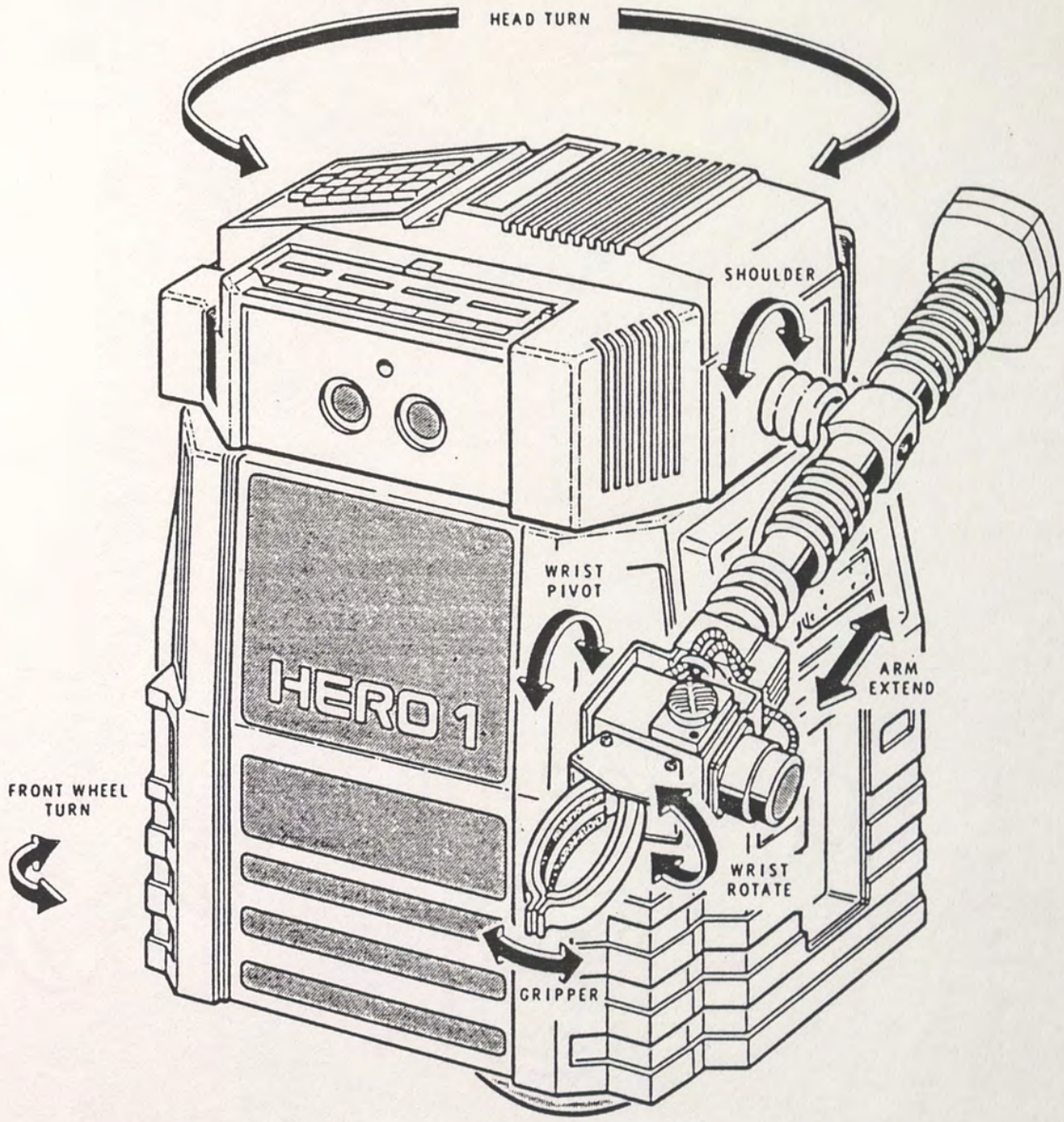

Figure 1. The Hero Robot.
Source: The Heath Company 1982

The availability of a high level language improves robot productivity by allowing programs to be written, debugged, and verified on a computer, offline from the robot. This program development process does not interrupt the robot's ongoing activities.

Overview of the Graphics Simulator

The Robot Interactive Graphics Simulator was developed to interpret commands written in the newly developed high level language and provide the programmer with a display of the motions which will be executed by the robot in response to these commands. This display gives the programmer visual verification of the commands in the program.

The graphics simulator provides the programmer with two views of the movements of the robot. The left side of the screen shows an overhead view of the robot's movements, while the right side of the screen shows a clearer view of the position of the arm, wrist, and gripper of the robot.

The graphics simulator interprets commands which are stored in a program file or typed in by the operator interactively. The graphics simulator can interpret the high level language commands with as little as one letter of the command or as much as the entire English word. If any errors exist in the high level language code, an error message which explicitly states the nature of the problem is written to the screen and to a log file. These features

help the novice programmer learn the Hero High Level Language and become accustomed to high level language programming systems available on industrial robots.

HERO PROGRAMMING LANGUAGE

The Heathkit Hero Robot is used primarily as an educational system which provides training in the principles of automation and robotics. The Hero is equipped with a microprocessor which must be programmed in one of two modes: the Learn Mode and the Program Mode (Heath Company 1982). In the Learn Mode, a teaching pendant is used to manually take the robot through a series of motions. These steps are simultaneously recorded in memory. The advantage of the Learn Mode is that no programming expertise is required in order to teach the robot to perform a specific operation. The disadvantage of the Learn Mode is that any robot production activities would have to be interrupted in order to teach the robot a new sequence of operations.

The Program Mode allows a series of machine language instructions to be manually entered into the microprocessor through the keyboard. This is a complex procedure which requires the programmer to keep track of microprocessor storage locations and their contents. The machine language statements consist of hexadecimal operation codes, which are difficult to remember. High level programming languages have been used since the 1950s to simplify computer programming (Hussain and Hussain 1981). High level languages overcome the drawbacks associated with machine

language programming because the computer keeps track of memory storage locations and contents. High level languages are easier to learn, write, and debug because the program statements are usually English words, mnemonics, or other familiar symbols.

Many high level languages have been written to simplify the task of programming industrial robots. Most industrial robots can be equipped with high level languages which improve programming efficiency and reduce the amount of time required to train robot operators. The high level language described herein was developed for the Hero Robot in order to overcome the down-time requirements of the learn mode and the programming difficulty of the Program Mode. The Hero High Level Language allows programs for the robot to be written and tested offline. The robot's production activities do not need to be suspended for long periods of time during reprogramming. The high level language capability also makes the Hero Robot more effective as an educational system because it gives the Hero a programming system which is more like the robots used in industry.

History of Robot Languages

Early robotics applications, such as spray painting and spot welding, used a simple teach and repeat programming method. This procedure was sufficient because a single task was simply repeated over and over. However, as robotics

applications increased in complexity, the need for more sophisticated programming methods became evident (Gruver, Soroka, Craig, and Turner 1983).

Early research on robot languages was performed at Stanford Artificial Intelligence Laboratory in the early 1970s. A language called WAVE was developed in 1973 which was successfully integrated with a vision system. In 1974, the Stanford group developed a second robot programming called AL. This language was used in programming assembly operations which used force sensing capabilities.

The VAL language was released in 1979 by Unimation. It was the first commercially available robot programming language. VAL is an extension of the BASIC high level language, and is supplied with the company's PUMA robots.

RPL is a FORTRAN-like language developed by SRI International to control the various machines contained in a robotic work-cell: manipulators, sensors, and machine tools.

In 1982, A Manufacturing Language (AML) was released by IBM. AML was designed to be used by operators with a wide range of programming expertise by providing a powerful base language with simple subsets. One version of AML which controls the IBM Model 7535 Robot can run on an IBM Personal Computer.

The HELP language was released in 1982 by General Electric Company. HELP is based roughly on PASCAL, and is

designed to be easy to learn and use. HELP supports simultaneous arm movement for multiple-arm robots (Bonner and Shin 1982).

SIGLA (SIGma LAnguage) is a new robot language developed by Olivetti for their SIGMA robots. SIGLA requires only 8K bytes of memory, yet allows parallel task execution to allow for multiple arms running together. (Salmon 1978). Until recently, these features were only available on research robotics applications.

Current research is being directed toward the development of task-level description languages for industrial robot applications (Gruver, Soroka, Craig, and Turner 1983). Task-level languages would allow operators to control the robot using statements such as "move object to machine A" without having to be concerned with those details necessary to control the robot.

Requirements for the Hero High Level Language

A series of requirements were established for the development of the high level language in order to ensure that the language would be an effective tool for simplified programming of the Hero Robot.

The first requirement for the high level language was that all the motors of the robot should be addressed. This included arm extension, shoulder rotation, gripper position, wrist pivoting and rotation, head rotation, body movement,

and steering wheel position. Some of the functions of the Hero Robot, such as motion and light detection, did not need to be addressed, but the structure of the language had to allow for expansion to include these functions at a later date. One unique feature of the Hero Robot is its speech capability. A requirement was established that the high level language address the speech function in a limited fashion.

The language had to be as easy to use as possible. This included the use of English words or mnemonics for all statements, commands, parameters, and options.

The statement parameters were required to be as consistent as possible between statements in order to reduce the time required to learn the language.

The Hero Robot Language System (including compiler, loader, and graphics simulator) was required to operate on an IBM Personal Computer.

The Hero Robot Language Interpreter

The Hero Robot is equipped with a special set of machine language instructions tailored to the robot's logical and mechanical requirements. These instructions provide direct control of motors, sensors, speech, and other operations which are unique to the robot. The internal program which contains these special commands is called the Robot Interpreter and the language used by the interpreter

is called Robot Language by the manufacturer (Heath Company 1983).

The subset of the Robot Language Interpreter Table which will be used by the Hero High Level Robot Language is presented in Table 1. The subset includes only the speech and motor commands.

TABLE 1

INTERPRETER TABLE

COMMAND	FORM	TITLE
71	71 MM MM	SPEAK, CONTINUE (EXTENDED)
72	72 MM MM	SPEAK, WAIT (EXTENDED)
C3	C3 SS XX	MOTOR MOVE, WAIT ABS (IMMEDIATE)
CC	CC SS XX	MOTOR MOVE, CONT ABS (IMMEDIATE)
D3	D3 SS XX	MOTOR MOVE, WAIT REL (IMMEDIATE)
DC	DC SS XX	MOTOR MOVE, CONT REL (IMMEDIATE)

NOTES: XX = distance, position, etc.
SS = select motor, speed, direction
MM = memory address

All the motor commands share several command mode parameters: timing (wait or continue), positional (absolute or relative), and speed (slow, medium, or fast). The speech command also uses the timing parameters, but not the positional or speed parameters.

The two timing modes are wait or continue. Wait type commands cause the robot to perform the stated command while

the rest of the program waits until the current operation is complete. Continue type commands cause the robot to start the stated command and continue with the rest of the program.

Positional modes are either absolute or relative. Relative commands tell the motor what direction and how far to go from the current position. Absolute commands tell the motor to go to a specific position relative to the coordinate system of the part.

Each motor can be driven at one of three speeds: slow, medium, or fast.

The Robot Interpreter will not allow more than one arm motor to be driven at one time. If attempted, the interpreter will drive one motor at a time, while subsequent moves wait their turn.

The Development of the High Level Language

The most direct approach for the development of a language for this robot was to translate the existing Robot Interpreter commands into an English command format which would be more understandable and easier to write.

The Hero High Level Robot Control Language utilizes the commands of the Hero's Robot Interpreter. The statements which are used in the High Level Robot Language are INITIALIZE, SPEAK, ARM EXTEND, ARM SHOULDER, DRIVE, GRIPPER, HEAD, TURN, WRIST PIVOT, and WRIST ROTATE.

The timing mode parameters (WAIT or CONTINUE), positional mode parameters (ABSOLUTE or RELATIVE), and the speed parameters (SLOW, MEDIUM, or FAST) were all translated to the English equivalents shown in parentheses. All of the move commands (except for the DRIVE command) are consistent in the required order of the mode parameters: a timing mode parameter, a positional mode parameter, and a speed parameter. The distance for the DRIVE command is always relative to the current position, so the positional mode parameter is not necessary.

The Hero Robot Technical Manual (Heath Company 1983) states that the motors may have difficulty starting up at the higher speeds (medium and fast). The manual suggests that the programmer can overcome this by starting the motor at slow speed for several units of distance, then run at medium speed for several units of distance, and then run at fast speed for the rest of the distance. The move commands in the high level language handle this "ramping" of the speed for the drive motor when compiled. Therefore, a single high level language DRIVE command may become three machine language drive commands when compiled.

The INITIALIZE command returns all the motors of the robot to the same locations as does an initialize command entered on the robot's keyboard. This gives the programmer the ability to start relative movements from a known position.

The Hero Robot has several preprogrammed speeches stored in read-only memory. In addition, several speeches were created using the language phonemes and are written into the Hero's random access memory by the compiler when those speeches are selected. The easiest way to specify a particular speech was to assign a "speech number" to each of the available speeches. The SPEAK command lets the programmer select from any of the available preprogrammed speeches by selecting the speech number.

The instructions written in the high level language are translated by a compiler into the robot interpreter machine language instructions in hexadecimal format. The machine language instructions can then be downloaded to the Hero by an RS232C interface which is available through the Heath Company.

Additions for Compiler and Graphics Simulator Statements

In addition to the commands to make the robot move and speak, the language had to contain some commands which would pass information to the compiler, loader, and graphics simulator. The commands which were added for this purpose are MEMORY, END, and commands starting with the phrase GRAPHICS.

When compiling a program for the Hero Robot, the compiler and loader need to have the address at which to start storing the program in the memory of the robot. The

MEMORY command lets the programmer set the address into which the next command will be entered. Once the memory address is set, the compiler increments the address according to the size of the statement which is entered until the end of the program or another MEMORY command is entered.

The END statement marks the end of the program. When either the compiler or the graphics interpreter reads an END statement, that program will end. If a program does not have an END statement, an error is generated by both the compiler and the graphics interpreter.

There are three statements to pass information to the graphics interpreter: GRAPHICS BACKGROUND, GRAPHICS LIMITS, and GRAPHICS STARTINGPOINT. These statements are ignored by the compiler. The BACKGROUND command lets the programmer specify a file which contains a drawing to be used as a background for the overhead view of the robot. The LIMITS command lets the programmer specify the limits of the viewing area displayed in the overhead view of the robot. The STARTINGPOINT command lets the programmer specify the initial position and direction of the robot with respect to the background.

A full description of all the high level language commands is provided in the Hero High Level Language Reference Manual.

Each of the statements of the High Level Language are described in this section. In describing the format of each statement, the following symbols are used:

* Capital letters represent the minimum required set of characters to allow recognition of a command. Lower-case letters indicate non-required characters.

* Phrases surrounded by < > indicate that the argument is required.

* Arguments separated by / indicate that only one of the listed arguments should be used.

* Statements may be entered in any combination of upper-case and lower-case letters.

The compiler and graphics simulator have other requirements regarding the source program.

* At least one space must be placed between the statement and each subsequent argument.

* The graphics simulator requires that exactly one complete command (command, mode parameters, and argument) be on each line of the source file.

* All non-motor commands (End, Initialize, Memory, Graphics, and Speak) must end in a semicolon (;).

* Any speed specified as fast or medium will be ramped up to that speed to avoid damage to the motors.

List of Commands

Arm Extend

Purpose. To control the extension of the arm.

Format. Arm Extend <Continue/Wait> <Absolute/Relative> <Slow/Medium/Fast> <inches>

Remarks. The speed may be specified as slow, medium, or fast. The inches parameter represents the amount of movement (non-negative for absolute movements; positive or negative for relative movements). In the absolute mode, 0 inches represents a fully retracted arm, and 5 inches is fully extended. The initialized position of the arm extend motor is fully retracted (0 inches).

Arm Shoulder

Purpose. To control the raising and lowering of the arm at the shoulder.

Format. Arm Shoulder <Continue/Wait> <Absolute/Relative> <Slow/Medium/Fast> <degrees>

Remarks. The shoulder motor raises and lowers the arm 150 degrees in the vertical plane. Zero degrees represents the fully lowered arm, and 150 degrees is fully raised. The initialized position of the shoulder is fully lowered (0 degrees).

Drive

Purpose. Instructs the robot to engage the drive motor and travel the specified distance.

Format. Drive <Continue/Wait> <Slow/Medium/Fast> <feet>

Remarks. Positive values cause the robot to go forward. Negative values cause the robot to go backward. All drive motions are relative to the current position.

End

Purpose. Marks the end of a Hero High Level Language program.

Format. End;

Remarks. An End statement must be the last line of every Robot Language program.

Graphics Background

Purpose. Adds the background in the specified file to the overhead view of the graphics simulator.

Format. GRAphics Background <OFF/filename.ext>;

Remarks. The file named must be in the Drawing Exchange Format specified by the software package AUTOCAD. This background may consist of only lines. The base reference point of the background drawing is the origin of the overhead view. The limits of the overhead view will expand to always include the full background, even if the limits are specifically set to less than the size of the background. This statement is ignored by the compiler.

Graphics Limits

Purpose. Sets the limits of the overhead view for the graphics simulator.

Format. GRAphics Limits <x1> <y1> <x2> <y2>;

Remarks. The parameters x1, y1, x2, and y2 are distances, measured in feet, from the origin (0,0 point) of the room. If a program does not specify any graphics limits, the default values are -3, -3, 3, 3. This statement is ignored by the compiler.

Graphics Reversearm

Purpose. Reverses the direction of arm movement.

Format. GRAphics Reversearm;

Remarks. The direction of arm movement depends on how the robot was assembled.

Graphics Startingpoint

Purpose. Sets the robot in the overhead view of the graphics simulator to the specified location in the "room" and at the specified angle.

Format. GRAphics Startingpoint <x> <y> <degrees>;

Remarks. The parameters x and y are distances in feet from the origin of the room. The angle is in degrees, and is positive in the clockwise direction. If a program does not specify a starting point, the robot will start from the 0,0 coordinate with an angle of 0 degrees. This statement is ignored by the compiler.

Gripper

Purpose. To instruct the robot to open and close the gripper.

Format. GRIpper <Continue/Wait> <Absolute/Relative> <Slow/Medium/Fast> <inches>

Remarks. The amount of movement is specified in inches. An absolute amount of 0 inches represents a closed gripper. The fully opened limit is 3.5 inches. The initialized position of the gripper is closed (0 inches). Note: the GRIPPER and GRAPHICS commands require a minimum of three characters of the command name to be distinguished from one another.

Head

Purpose. To rotate the head of the robot.

Format. Head <Continue/Wait> <Absolute/Relative> <Slow/Medium/Fast> <degrees>

Remarks. In the absolute mode, zero degrees represents full counterclockwise rotation of the head. 350 degrees represents full clockwise rotation of the head. The initialized position of the robot's head is in the center position (approximately 175 degrees).

Initialize

Purpose. Tells the robot to move all motors to specific home positions.

Format. Initialize;

Remarks. Initialize moves all motors to the same positions as the initialize code which is entered from the robot's keyboard. This command can be used to reinitialize the position of the robot motors when moving from one program to another. The initialized positions of each motor are described in the Remarks section of the respective motor.

Memory

Purpose. Sets the address of the robot to the address specified.

Format. Memory <memory address>;

Remarks. The memory address may be written in decimal or hexadecimal format. If written in hexadecimal format, the address must be preceded by a $, such as $0100. If no MEMORY command is specified in a program, the default is $0040 (64). A MEMORY statement would typically be the first statement in a program. The lowest allowable memory address is $0039 (63). The highest allowable memory address is $0C50 (3152). If an invalid memory location is specified, a default value of $0100 (256) is used as the starting address. The MEMORY command must be terminated with a semicolon.

Speak

Purpose. Instructs the robot to say the specified phrase.

Format. Speak <Continue/Wait> <speech number>;

Remarks. The 19 speeches are as follows:

1: "HELLO, MY NAME IS HERO."
2: "HELLO. I AM HERO. THE HEATH EDUCATIONAL ROBOT."
3: "I CAN TALK, LIKE THIS."
4: "I CAN MOVE MY ARM."
5: "I CAN USE MY GRIPPER."
6: "I CAN TURN MY HEAD."
7: "AND I CAN MOVE ABOUT."
8: "I HAVE A BRAIN. JUST LIKE YOU DO. BUT MY BRAIN IS A COMPUTER. MY OWNER PROGRAMS MY COMPUTER FOR ME AND I ALWAYS DO AS I'M PROGRAMMED."
9: "THERE IS NO SUCH THING AS A BAD ROBOT. JUST A MISPROGRAMMED ONE."
10: "GOSH, I THINK I'M JUST ABOUT PERFECT!"
11: "I THINK YOU ARE CUTE. GIVE ME A HUG. ROBOTS NEED LOVE TOO."
12: "YOU ARE VERY ATTRACTIVE FOR A HUMAN."
13: "YOUR WISH IS MY COMMAND."
14: "OH MY! PLEASE DO NOT DO ANYTHING TO HURT ME."
15: "OH NO! I DO NOT DO WINDOWS."
16: "I WILL NOW RUN THE PROGRAM YOU HAVE WRITTEN IN MY NEW HIGH LEVEL LANGUAGE."
17: "I HAVE COMPLETED THE PROGRAM AND WILL NOW RETURN TO EXECUTIVE MODE."
18: "I AM A HANDSOME ROBOT, BUT THE RB5X IS UGLY."
19: "I LIKE TALKING TO THE IBM PC. IT HAS NICE PERIPHERALS."

Speeches 1 through 15 were written by Heathkit for the Hero Robot and are stored in ROM. Speeches 16 through 19 were composed using the speech dictionary provided with the robot. Additional speeches may be added by modifying the compiler.

Turn

Purpose. To steer the robot in the desired direction.

Format. Turn <Continue/Wait> <Absolute/Relative>
<Slow/Medium/Fast> <degrees>

Remarks. Straight ahead is 0 degrees, full left is -90 degrees, and full right is 90 degrees in absolute terms. The initialized position of the steering motor is 0 degrees (straight ahead).

Wrist Pivot

Purpose. To control the pivot operations of the gripper.

Format. Wrist Pivot <Continue/Wait> <Absolute/Relative>
<Slow/Medium/Fast> <degrees>

Remarks. The wrist pivot motor moves the gripper a total of 180 degrees (90 degrees above and 90 degrees below the axis of the arm). For absolute movements, a value of -90 degrees represents 90 degrees below the arm axis. The initialized position of the wrist pivot is all the way up (90 degrees).

Wrist Rotate

Purpose. To control the rotate operations of the gripper.

Format. Wrist Rotate <Continue/Wait> <Absolute/Relative> <Slow/Medium/Fast> <degrees>

Remarks. The wrist rotate motor rotates the gripper through 350 degrees. Zero degrees represents the full counterclockwise position of the wrist. The initialized position of the wrist is 183 degrees. Note: the Hero User's Guide states that the initialized position is the full counterclockwise limit (0 degrees); however, the two Hero Robots used in the development of the high level language both initialize to 183 degrees.

GRAPHICS SIMULATOR DESIGN

Computer graphics are widely used in industrial simulation and training systems. Numerical control programming languages, such as APT, usually include the ability to verify the NC program by plotting the tool path on a graphics plotter or graphics display. Many of the advancements in hardware and software for computer graphics have resulted from research for designing real-time computer graphic simulators for training aircraft pilots.

Many industrial robots have graphics simulator tools available for their high level languages. The simulator allows the programmer to verify that the robot will move as expected in response to the program before downloading the program to the robot.

The Robot Interactive Graphics Simulator was developed for the Hero Robot in conjunction with the High Level Language. The availability of a graphics simulator as a training and programming tool lets operators and programmers verify their programs in much the same manner as with an industrial robot.

Requirements of the Graphics Simulator

A series of requirements were established for the development of the graphics simulator in order to ensure

that the simulator would be an effective tool for displaying robot movements from a high level language program.

The primary requirement was that the graphics simulator had to accurately depict the movements of the robot when executing high level language instructions.

Another requirement for the graphics simulator was that two views of the robot would be shown on the screen at the same time. One view of the robot would be an overhead view, which would show the position of the robot relative to the surroundings. The second view of the robot would be a view which would more clearly show the position of the arm, wrist, and gripper of the robot.

A status line displaying the current position of all the movable robot parts was required to be displayed on the screen. The absolute position of the parts should be displayed in the units used by the command language.

The graphics simulator had to include an interpreter to check the syntax of programs entered by the user. The interpreter would execute only the correct high level language instructions.

The graphics simulator had to use color graphics and had to run on an IBM Personal Computer.

The final requirement of the graphics simulator was that the views of the robot had to have hidden lines removed.

Limitations for the Graphics Simulator

Two limitations for the graphics simulator had to be set early in the design stage. The first limitation was to allow the robot to move only forward and backward. The second limitation was to prohibit CONTINUE timing modes.

The Hero Robot is a free-standing robot which moves around on three wheels. The front wheel of the robot can turn 90 degrees from the straight ahead position in either direction, thereby steering the robot. The front wheel is also the wheel that drives the robot and can move the robot either backward or forward. The two rear wheels of the robot basically follow the front wheel. When the front wheel is turned off of center, it will remain at its new position until another command turns it to a different position, causing the robot to move in an arc. The overall movement of the robot is analogous to that of a tricycle.

Most industrial robots on the market today are either fixed in their position or able to move only forward and backward. The analysis of the movement of the robot when the front wheel is turned would have complicated the analysis of motion greatly. Since industrial robots do not typically have this type of motion, the graphics simulator was limited to only recognizing driving motion when the steering wheel is at 0 degrees (pointing straight ahead). If a DRIVE command is received when the steering wheel is at any position other than 0 degrees, an error message is

generated and the DRIVE command is executed as if the wheel was set to 0 degrees.

Timing mode parameters for the Hero Robot can be either CONTINUE or WAIT for any motor move or speech command. In the WAIT mode, the motor move or speech command must finish executing before another command is executed, therefore limiting the robot to one command at a time. In the CONTINUE mode, the robot can drive forward and move any one of the other motors at the same time or say a speech while moving. When one command (either the drive command or the other motor command) finishes, the next command is executed.

For the graphics simulator to correctly analyze the CONTINUE timing mode of the high level language, the kinematic equations of every motor of the robot would have to be known in order to determine which command would finish executing first. Since there are three speeds for every motor, this would introduce many variables into the system. The graphics simulator was limited to recognizing only the WAIT timing mode parameter. If a CONTINUE mode timing parameter is included in a program, an error message is generated and the command is executed as a WAIT mode timing parameter.

Theory of 3D Data Manipulation

This section is a summary of the theories of manipulation of three-dimensional data using matrix methods.

A knowledge of matrix algebra and trigonometry is assumed. A more complete description of the concept of representation and manipulation of 3D data using matrix methods is available from Foley and Van Dam (1982), Artwick (1984), and Angell (1981).

A point named P in three-dimensional (3D) space can be referred to by its cartesian coordinates, P(x,y,z), or can be placed in vector format, such as

$$P = [x\ y\ z].$$

In order to represent the translation of the data point P to a new location, P', which is Dx units in the X-direction, Dy units in the Y-direction, and Dz units in the Z-direction we would write in vector format

$$P' = [x'\ y'\ z'] = [x\ y\ z] + [Dx\ Dy\ Dz] = P + T.$$

Rotation of a matrix point around the Z-axis by A degrees is defined mathematically as

$$\begin{aligned} x' &= x * \cos(A) - y * \sin(A), \\ y' &= x * \sin(A) + y * \cos(A), \\ z' &= z. \end{aligned}$$

In matrix form, this can be represented by the point P by a three-by-three matrix, R, such that

$$P' = P * R = [x\ y\ z] \begin{vmatrix} \cos(A) & \sin(A) & 0 \\ -\sin(A) & \cos(A) & 0 \\ 0 & 0 & 1 \end{vmatrix}.$$

This three-by-three form of the rotation matrix does not allow for any translation of the data point at the same time.

Rotation and translation of a data point can be combined by expressing the coordinate data as homogeneous coordinates, which can be simply thought of as adding a fourth dimension of 1 to the vector point. Therefore, the point P would now be represented as

$$P = [x\ y\ z\ 1].$$

A matrix T which would translate P to point P' as described above would become the four-by-four matrix

$$T(Dx,Dy,Dz) = \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ Dx & Dy & Dz & 1 \end{vmatrix},$$

so that the equation for the translation is now

$$P' = P * T.$$

The equation for a point which is rotated around the z-axis then translated by (Dx,Dy,Dz) would be

$$P' = P * T = [x\ y\ z\ 1] \begin{vmatrix} \cos(A) & \sin(A) & 0 & 0 \\ -\sin(A) & \cos(A) & 0 & 0 \\ 0 & 0 & 0 & 0 \\ Dx & Dy & Dz & 1 \end{vmatrix}.$$

The matrix for an X-axis rotation is

$$Rx(A) = \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(A) & \sin(A) & 0 \\ 0 & -\sin(A) & \cos(A) & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}.$$

The matrix for a Y-axis rotation is

$$Ry(A) = \begin{vmatrix} \cos(A) & 0 & -\sin(A) & 0 \\ 0 & 1 & 0 & 0 \\ \sin(A) & 0 & \cos(A) & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}.$$

For all of these transformation matrices there exists another transformation matrix which will reverse the effect

of the original transformation. The inverse of a translation matrix is obtained by negating Dx, Dy, and Dz. The inverse of each of the three rotation matrices is obtained by negating the angle of rotation. If a transformation matrix Tfm is made up of a series of elementary transformation matrices T1,T2,...Tn such that

$$Tfm = T1 * T2 * \ldots * Tn,$$

then the inverse transformation InvTfm is written as

$$InvTfm = InvTn * InvT(n-1) * \ldots * InvT2 * InvT1.$$

Note that the order of the inverse transformation matrices is the reverse of the order of the transformation matrices, and that multiplication of two matrices is not always commutative.

A series of various rotations and translations can be reduced to a single transformation matrix, Tfm, of the form

$$Tfm = \begin{vmatrix} r11 & r12 & r13 & 0 \\ r21 & r22 & r23 & 0 \\ r31 & r32 & r33 & 0 \\ tx & ty & tz & 1 \end{vmatrix}.$$

This method of representing coordinate data and transformation matrices is used with not only computer graphics but also many other engineering applications, such as numerical control. The data point arrays used throughout the graphics simulator are all represented as homogeneous coordinates as described in the preceding paragraphs. All of the transform matrices used in the graphics simulator are 4x4 matrices, as described in the preceding paragraphs.

Theory of Hidden Surface Elimination

In order to achieve the most realistic view of a three-dimensional object on a computer, solid surfaces of the object must be depicted as blocking the view of the surfaces behind them. Unfortunately, the simplest methods of displaying graphical data on a computer (drawing of points and lines) do not incorporate any removal of hidden lines. Special techniques must be used to remove hidden lines and surfaces when displaying graphical data on a computer.

Hidden surface techniques vary in complexity, speed, and memory requirements. Since the graphics simulator had to run on an IBM Personal Computer with 256 kilobytes of memory, the memory can be viewed as limited when compared to systems with megabytes of memory available. The speed of a personal computer also limited the choice of techniques to those which would execute quickly.

Depth-Sort algorithms and Z-Buffer algorithms (Foley and Van Dam 1982) will produce very accurate displays of information, but require a great deal of memory and would take too much time on a personal computer. These algorithms were not used in the graphics simulator.

A technique of hidden surface elimination which is fast and does not require a great deal of memory is the technique called Backside Elimination (Artwick 1984). A viewer can see a surface plane of an object if the vector normal (a perpendicular line) to that surface plane is pointing toward

the viewer, and cannot see those surfaces for which the normal points away from the viewer. By eliminating the surfaces which point away from the viewer (those with their backsides toward the viewer), the total number of surfaces is greatly reduced, typically by 50% or more. This also reduces the number of surfaces to which other hidden surface techniques may be applied.

The direction of the normal for a plane may be determined from the plane equation of the plane. The equation describing a plane can be determined by using three vertices of the plane. The form of the equation of any plane is

$$Ax + By + Cz + D = 0$$

and the equation of the normal to that plane is the vector with the (x, y, z) coordinate values of (A, B, C). If an object lies on the positive Z-axis and the viewer is at the origin, then any plane with a normal which has a negative z-value (C) is facing the viewer and can be seen. Similarly, any plane with a positive z-value has its backside toward the viewer and cannot be seen. This algorithm is used within the graphics simulator to determine which planes can be seen.

Backside elimination is useful for removing hidden lines from a single convex polyhedron. However, if more than one object is being plotted or if the object is non-convex, you may have one surface partially or completely blocking another surface, and other hidden line removal

techniques must be applied. Since the robot arm could block the view of the robot body, and vice-versa, additional hidden surface elimination techniques are needed for use in the graphics simulator.

One object blocks the view of another object if it is closer to the viewer and in the same line of sight. If the farther surface is plotted first and then the closer surface is plotted and filled in, the closer surface will block the farther surface and the view will be correct in many cases. This technique is called a Priority Sort (Artwick 1984). In the graphics simulator, the vertex with the smallest Z-value is determined for each plane. The planes are then sorted by these minimum-Z-values, and the planes are plotted in a decreasing Z order. There are cases where the priority sort method will not produce accurate results (such as when two surfaces are intertwined), but these cases did not occur in the graphics simulator.

According to Artwick (1984), the combination of a back-side elimination and a priority sort is the most time-efficient and memory-efficient method of hidden surface elimination for objects consisting of multiple polyhedrons. These methods are used in the graphics simulator for hidden surface removal.

Data Structure for 3D Data

After the concepts for the manipulation of data and removal of hidden surfaces were solidified, the structure of the graphical data had to created. It was necessary to create the data structure before digitizing the part data to ensure that all of the necessary information would be recorded.

In order to draw the robot, points and lines would be required. Each point had to be a four-dimensional coordinate consisting of the X, Y, and Z data values and a fourth value of 1 required by the matrices manipulations (see the section titled "Theory of 3D Data Manipulation"). Therefore, an array called "Points" was created, where the index of the array consisted of four real numbers. Each line must consist of two points. Since the point data was already stored in the array "Points," the line information could consist of two integers indicating which two points were the end points of the lines. Therefore, an array "Lines," consisting of the two integer indices, was created.

The hidden surface elimination method outlined previously requires that, for each plane, the vertices of the plane had to be known. Similar to the "Lines" array, this information could consist of pointers to values in the array "Points." It was recognized that it might be necessary to know which lines made up the plane. This information could consist of integer pointers to values in the

array "Lines" for each of the four edges. Therefore, an array "Planes" was created where each value of the array consisted of a record of the "Vertices" (the four integer pointers to the array "Points") and the "Edges" (the four integer pointers to the array "Lines").

When calculating new positions of graphic objects, one can choose to have the program work in either a relative mode or in an absolute mode. In a relative mode, the points are updated from the last position. In an absolute mode, the current values of points are calculated by transforming the original data points. Relative mode is easier for simple translations, but if an object must be rotated around a point other than the origin, the object would first have to be translated, then rotated, then translated again. The additional steps could end up causing a build-up of errors and is generally more complex. It was therefore decided to work in an absolute mode.

Since working with absolute data requires access to the original data, the array "Points" had to be left intact. A second array, "XfmdPts," was required to store the transformed values of the array "Points." Like the array "Points," each index of this array had to consist of four real numbers.

This is the general description of the graphical data which was defined before the data was digitized. Further details of the data structure are presented in the section

of the thesis titled "Development of the Graphics Simulator Software."

Digitizing the Part Data

In order to display and manipulate an object with a computer, the object must be described as a series of X, Y, and Z coordinate points, lines, and planes. Digitizing is the act of converting a 2D drawing or a 3D object into a series of points, lines, and planes.

The process of digitizing should not be taken lightly. Many of the decisions made during the digitizing greatly affected the overall graphics simulator program. By carefully planning the digitizing and considering the information and the format that might be needed later in the program, the overall program design was simplified. If this had not been as carefully executed, the design could have been complicated.

The digitizing of the Hero Robot consisted of four tasks: dividing the description of the robot into the moving parts, establishing the coordinates for each robot part, measuring and recording the data, and verifying the data. Each of these tasks is described in detail in the following subsections.

Dividing the Robot into Moving Parts

The movements of the Hero Robot are associated with specific parts of the robot. The moving parts of the robot

are as follows: the steering wheel, the robot body, the head, the rotating arm, the arm extension, the pivoting wrist, the rotating wrist, and the grippers. Each of these moving parts was digitized separately. This made the task of keeping track of the points located in various sections of the robot much easier. Also, each individual robot part could be viewed separately, allowing for easier visual verification (debugging) of small sections of the robot.

Since the values of the data in the arrays "Lines" and "Planes" were pointers to the indices of the array "Points," the addition or deletion of one point would affect the indices of all the points following it, thereby affecting the values in the arrays "Lines" and "Planes." By separating the data in the digitizing process, the effect of modifying the point data was localized to the one part.

The data associated with the separate parts are combined into the arrays "Points," "Lines," and "Planes" when the data is read from the data files as the program is executed. The program corrects the values in the "Lines" and "Planes" arrays for the new indices of the "Points" array as the data is being read from the data file.

Establishing the Coordinates for Each Robot Part

Each of the moving parts of the robot has a motion associated with it: the head rotates, the wrist pivots, and so forth. A motion of one of the robot parts would affect

not only that part but also the parts which are dependent upon the part in motion. For example, rotating the arm at the shoulder affects the position of the arm, wrist, and each gripper claw, but does not affect the position of the head.

Since each part has a motion associated with it, in order to transform the points separately, the points associated with each part would have to be transformed separately. Each part would therefore require its own transformation matrix. Therefore, each point could also have its own coordinate system. The transformation matrix for the part could include not only the trasformation of the part but also a transformation to another coordinate system which would be compatable with the other parts.

As discussed previously, points which are rotated using matrix transformations can be rotated only around a specific axis or set of axes. Since it was already decided to use an absolute coordinate system, it made sense to establish the coordinate system in such a way that the axis around which the part would be rotated would pass through the point of rotation. This would make it easy to rotate the part because the part does not need to be translated before it is rotated. A secondary consideration in establishing the coordinate data was to make it convenient to digitize the data by taking advantage of the symmetry of some of the robot parts.

The robot coordinate system is the system to which all of the robot parts are transformed. This coordinate system is set up such that the origin is at the center point of the front wheel (the steering wheel) of the robot at the point where the wheel touches the floor. The axes are aligned with the X-axis pointing out of the front of the robot, the Y-axis pointing out the left of the robot, and the Z-axis pointing straight up through the robot. This coordinate system was easily transferred to the both the overhead view and the shoulder view of the robot.

Measuring and Recording the Part Data

The points, lines, and planes of the Hero Robot were carefully measured and recorded. Although recording the point and line data was straightforward, the plane data was slightly more complex.

The hidden surface elimination routines described in a previous section require knowing what points make up the vertices of the plane. The vertices are used to determine the plane equation and the direction of the normal to the plane. The order in which the points are recorded determine whether the normal of the plane will point out of the robot or into the robot. The vertices were recorded in a counter-clockwise direction with the reference being the outside of the robot for every plane. Although it was not required, the "Edge" information for the array "Planes" was also

recorded in a counterclockwise direction and in the same order as the vertices.

The robot was divided into planes consisting of three or four vertices. If a plane consisted of only three vertices, the values of the indices to the array "Points" for each of the vertices was stored in the first three values of "Planes.Vertices" and a zero was stored in the fourth value. The fourth value of the "Planes.Edges" data was also set to zero for planes with only three edges.

Verifying the Part Data

A utility program was written to aid in debugging the part data. The program, named VIEWPART, would read the part data from any of the part data files and plot the data to the screen. The operator could then rotate the part by specifying the angles of rotation around the three axes to the program. This visual verification was very useful in debugging the data for each part, and some of the routines written for this utility program were later used for the graphics simulator.

Programming Language Selection

One of the requirements of the high level language was that the system had to operate on an IBM Personal Computer. This limited the selection of the programming language to BASIC, FORTRAN, and PASCAL, as these were the only high level languages available. FORTRAN was eliminated because

it did not have any graphics capabilities. The interpretive BASIC language, BASICA, had all of the necessary graphics commands with which to write the graphics simulator, but since it is an interpretive language it would execute at an unacceptably slow speed. The only version of a compiled BASIC language available to the author would not compile the necessary graphics commands.

TURBO Pascal Version 3.0 by Borland International is an enhanced version of standard PASCAL which runs on a wide variety of microcomputers including the IBM Personal Computer. TURBO Pascal is an inexpensive product which compiles and executes very rapidly. TURBO Pascal has good string handling capabilities which were needed for the interpreter section of the graphics simulator. In addition, TURBO Pascal includes as an extension to the PASCAL language a set of graphics commands for the IBM Personal Computer. TURBO Pascal was selected as the programming language not only for the Robot Interactive Graphics Simulator but also for the rest of the programs in the HICLASS system.

DEVELOPMENT OF THE GRAPHICS SIMULATOR SOFTWARE

The Robot Interactive Graphics Simulator was developed using top-down programming design techniques (Gilbert 1983). Top-down design means that the program is written in layers, starting with the main program and proceeding to the routines called by each routine, until the entire program is completed. Library routines which have already been written and debugged are included as needed.

Each routine was originally created as a "stub program" (Gilbert 1983), a blank shell containing only statements to indicate that the routine had been entered and exited. This way the flow of the program was tested and debugged before the code was created. The procedures Entering and Exiting were called for this purpose throughout the program. These routines write a comment containing the procedure name to the list device if the Boolean variable Debug_Flow is true.

Throughout the development process, the program was tested at the completion of each layer. The program was maintained in an executable state. Debugging the program early in the development cycle ensured that the flow of the program and the foundation routines were correct and greatly reduced the time required for debugging the software as new sections of the program were added.

Each routine in the program was designed to have one and only one entry point and exit point. The program was designed to capture errors. When an error occurs, the program will acknowledge the error and either proceed or end gracefully rather than crashing. If an abortive error occurs, the Boolean variable Aborting is set to true and each routine will then start returning to the calling procedure until the main program finishes executing.

The following sections describe the graphics simulator program. The first section describes the flow of the higher level routines. The second section concentrates on the code for manipulating and displaying the graphical data. The third section describes the command interpreter. The routines described within each section are listed within the section, so a knowledge of PASCAL is assumed. The complete program listing of the Robot Interactive Graphics Simulator is in Appendix B.

Overview of the Program Flow

This section describes the general program flow of the graphics simulator. The routines are described from the top-level main program down to the point where the program branches off into the graphics manipulation routines and the command interpreter routines, which are described in the following sections.

The flow of the main routine of the program is as follows.

```
begin { Main Program }
  Initialize;
  if not Aborting then ReadAndDraw;
  WrapUp;
end.
```

The program is initialized in the procedure named Initialize. The procedure ReadAndDraw, containing the command interpreter and graphics manipulation is executed if the initialization is successful. At the end of the program, the procedure WrapUp is executed to end the program properly.

The flow of the initialization routine Initialize is as follows.

```
begin  { Initialize }
  InitVariables;
  Welcome;
  ReadDataFiles;
  if not Aborting then ReadFontFile;
  if not Aborting then GetSourceName;
  if not Aborting then StartLogFile;
  if SaveTokens and not Aborting
    then StartTokenFile;
  Exiting('Initialize');
end;
```

The procedure InitVariables is called to initialize some of the variables used throughout the program, such as the Boolean flag Debug_Flow described earlier and the limits of the robot movement.

The procedure Welcome writes a welcome message to the operator. The welcome message contains the name and version number of the program and the name of the author.

The data files containing the point, line, and plane data are read by the procedure ReadDataFiles. If an error occurs while reading any of the data files, the Boolean variable Aborting is set to true and the remaining part data files will not be read. The robot part data files must be on the logged disk drive in a subdirectory called \DATA.

If no error occurred while reading the parts data file, the procedure ReadFontFile is called. ReadFontFile reads a data file, named "4x6.FON," containing the character set for a graphics font. The graphics font is used to write text to the graphics screen. This font, four pixels by six pixels for each character, is much smaller than the standard text font and will fit 64 characters across the low resolution graphics screen. If an error occurs while reading the font data, the flag Aborting is set true.

If the part data files and the font file were read correctly, the procedure GetSourceName is called. GetSourceName determines the name of the file containing the high level language source program and opens that data file. The source file name is typically passed as the first parameter in the run-string of the program. If the run-string does not contain the name of the source file or an error occurs while opening that file, the operator is prompted for the name of the source file.

The next procedure, StartLogFile, will open a log file into which the graphics simulator writes every high level

language command executed and every error message. The log file will be given the name of the source file with the file extension ".LOG" added to the name. If an error occurs while opening this file, a message is displayed, the Boolean variable LogCommands is set to false, and the program proceeds without logging commands or error messages.

The procedure StartTokenFile will be called if the Boolean variable SaveTokens was set to true in the procedure InitVariables when the program was last compiled. This procedure opens a data file into which the graphics simulator stores the tokens from each command interpreted. The token file is given the name of the source file with the extension ".GTK" added to the name. This is the last procedure called within the Initialization procedure.

The flow of WrapUp is as follows.

```
begin
  Entering('WrapUp');
  if SourceOpen then close(Source);
  if (SaveTokens and TokenOpen) then close(TokFile);
  if LogCommands then
    begin
      writeln(LogFile);
      writeln(LogFile,ErrorCount:5,
                              ' Errors Encountered.');
      close(LogFile)
    end;
  if not Aborting
    then TextMode(C80) ;
  TextColor(LightGray) ;
  CrtExit ;
  LowVideo ;
  Exiting('WrapUp');
end;
```

WrapUp prepares for ending the program. The source, token, and log files are closed if they were opened. The screen is restored to the text mode with the text color set to light gray. If the program is not ending in an abort condition, the screen is cleared.

The procedure ReadAndDraw calls the routines which deal with command interpretation and the manipulation and plotting of the 3D data. The graphics manipulation routines were developed and fully debugged before the command interpretation routines were developed. Although this method is contrary to the principles of top-down design, this method was necessary for proper debugging. If the two sections would have been developed together, it would have been difficult to tell if an error was caused by the command interpreter or the graphics manipulation.

The program listing of the procedure ReadAndDraw is as follows.

```
begin
  Entering('ReadAndDraw');
  RandDinit;
  ScreenInit;
  while not (Aborting or Done) do
    begin
      UpdateStatusLine;
      TransformPoints;
      DrawOverheadView;
      DrawShoulderView;
      ReadSourceCode;
    end;
  Exiting('ReadAndDraw');
end;
```

The procedure RandDinit initializes some of the program variables used in ReadAndDraw.

ScreenInit performs the graphics screen initialization for the graphics simulator. The display is set for low resolution color graphics. The color palette is selected as cyan, magenta, and white on a black background. Four graphic windows are then declared, with window 1 as the overhead view, window 2 as the shoulder view, window 3 as the prompt display area, and window 4 as the status line display area. Borders are drawn around windows 1 and 2.

The hidden surface algorithm required that the planes be filled in as they were plotted. The TURBO Pascal command for area fills requires at least three colors. Since the high resolution mode on the IBM Personal Computer allows only one color and a black background, the low resolution mode had to be used for the graphics simulator.

The ReadAndDraw routine then enters a loop in which five procedures are repeated until either Aborting or Done is set to true (Aborting and Done are both Boolean variables). The five procedures are UpdateStatusLine, TransformPoints, DrawOverheadView, DrawShoulderView, and ReadSourceCode.

UpdateStatusLine updates the status line shown in window 4. The status line shows the current position of each of the robot parts. Table 2 shows the robot parts and units for each of the abbreviations used in the status line.

TABLE 2

STATUS LINE DEFINITION

STATUS LINE ABBREVIATION	ROBOT PARTS	UNITS
X	Body Position(X)	feet
Y	Body Position(Y)	feet
ST	Steering	degrees
BD	Body	degrees
HD	Head *	degrees
SH	Arm Shoulder	degrees
AX	Arm Extension	inches
WP	Wrist Pivot	degrees
WR	Wrist Rotate	degrees
GR	Gripper	inches

* Note that the angle of the head is relative to the current body angle.

The procedures TransformPoints, DrawOverheadView, and DrawShoulderView perform the graphics manipulation and display. These procedures are discussed in the next section. The procedure ReadSourceCode reads and interprets the next high level language instruction. ReadSourceCode is discussed in the section titled, "Overview of the Command Interpreter Software."

The process of updating the status line, transforming the data points, drawing the two robot views, and interpreting the next high level language command are repeated until an abortive error occurs, the operator elects to exit the program, or an END command is interpreted. If an abortive error occurs, the Boolean variable Aborting is set true and the loop ends. If any of the other options

occur, the Boolean variable Done is set true and the loop ends.

After the end of the loop, the ReadAndDraw routine ends and returns to the main routine which executes the WrapUp procedure. The program stops after the WrapUp procedure returns to the main routine.

Overview of the Graphics Software

The Robot Interactive Graphics Simulator software has three procedures which control the manipulation and display of the graphics data: TransformPoints, DrawOverheadView, and DrawShoulderView. These three routines are contained within the procedure ReadAndDraw. TransformPoints sets up the transform matrices and multiplies the original part data points by the transform matrices according to flags set by the interpreter. DrawOverheadView transforms the points for the view and draws the overhead view. DrawShoulderView transforms the points for the shoulder view and draws the shoulder view.

The robot is divided into nine different parts: the steering wheel, the body, the head, the rotating arm, the extended arm, the pivoting wrist, the rotating wrist, and the two gripper claws (gripp and gripm). As discussed in the section titled "Digitizing the Robot Parts," each part can have its own coordinate system and must therefore have its own transform matrix.

The position of some parts affect the position of other parts. For example, the position of arm shoulder affects the position of the extended arm, the wrist pivot motor, the wrist rotate motor, and the grippers. Also, the position of some parts is not affected by the position of other parts. For example, the position of the head is not affected by the position of the arm. An array of the original data points, "Points," and an array of the transformed points, "XfmdPts," will both be maintained in the program. If a part has not been moved and is not affected by another part which has been moved, it is not necessary to recalculate the transform matrix or to transform the points for that part since the old values still exist in the array "XfmdPts." This method is used within the graphics simulator, and does not waste any computational time.

If the transformation matrices are created such that the transform matrix for a part transforms the coordinate system of the part to the coordinate system of the part that affects it, the two transform matrices can be multiplied to form another transform matrix. All of the transform matrices can build on the transform matrix of the controlling part to form transform matrices which will put the points into the final robot coordinate system. This algorithm is used for constructing the transform matrices in the graphics simulator.

A record of variables is maintained for each part. The array of records, named "Parts," is declared as follows.

```
Coordinate = array[1..4] of real ;
RobotParts = (Steering, Body, Head, Arm,
              ExtendArm, WrPvt, WrRot,
              Gripp,Gripm);
IndexRec = record
             First,
             Last: integer;
           end;
TransformMatrix = array[1..4] of Coordinate;
RobotRec = record
             Index: array[1..3] of IndexRec;
             Position: real;
             Absolutes: array[1..2] of real;
             RecalculateFlag: boolean;
             T: TransformMatrix;
           end;
Parts: array[RobotParts] of RobotRec;
```

The Boolean variable RecalculateFlag is set true if the transform matrix and data points need to be recalculated for the part. The values within the integer array "Index" are the values of the starting and ending indices within the arrays "Points," "Lines," and "Planes" for the part data. The current position of the part is stored in "Position." The transform matrix for the part is stored in "T."

The flow of procedure TransformPoints is as follows.

```
begin
  DrawText(1,'TRANSFORMING COORDINATES',Yes);
  for RbtPrt:=Steering to Gripm do
    with Parts[RbtPrt] do
      if RecalculateFlag then
        begin
          CalculateTMatrix;
          CalculateTfmMatrix;
          with Index[1] do
            MatMult(First, Last);
        end;
end;
```

The DrawText procedure draws the string passed to graphics window 3 in the 4x6 font.

Next, a loop is started in which the RecalculateFlag is checked for every part in the robot. If the value of RecalculateFlag for that part is set true three routines are called: CalculateTMatrix, CalculateTfmMatrix, and MatMult. These routines will only be called if the part was moved or was affected by another part being moved.

The procedure CalculateTMatrix will calculate the transform matrix "T" for the part. The "T" matrix will transform the points of that part to the coordinate system for that part. This value of the matrix "T" is temporary.

The procedure CalculateTfmMatrix will multiply the transform matrix for the part by the transform matrix of the part that affects it. This new matrix is stored in matrix "T" and in the matrix "Tfm." This value of the matrix "T" will be used to determine the transform matrix of the part that the current part affects. This matrix will transform the data points to the robot coordinate system.

The procedure MatMult will multiply the original data points for the part by the transform matrix "Tfm" and store these values in the array "XfmdPts." All the points in the array "XfmdPts" are in the robot coordinate system.

The points in the array "XfmdPts" must be transformed from the robot coordinate system to the coordinate system for the overhead view and to the coordinate system of the

shoulder view. A variable "ViewPts" was declared as follows.

```
PointType = array[MaxPoints] of Coordinate;
ViewType = (Overhead, Shoulder);
ViewRec = record
            Vpts: PointType;
          end;
ViewPts: array[ViewType] of ViewRec;
View: ViewType;
```

This creates a set of points "Vpts" for each view and a variable named "View" which can have the values Overhead or Shoulder. A set of points for each view was not required, but the memory was available for two arrays so two arrays were created. When the variable "View" is set to the current view, then any procedure referencing the variable "ViewPts[View]" references the set of points "Vpts" for the current view. A single set of graphics utilities which reference the points in this manner were written and used for plotting each of the views. These utilities were FindHiddenLines, SortPlanes, MaxMin, Scale, and PlotPlanes.

The listing of the procedure DrawOverheadView is as follows.

```
begin
  Entering('DrawOverheadView');
  View:= Overhead;
  OverheadViewTransform;
  FindHiddenLines ;
  SortPlanes;
  MaxMin ;
  TextXLcl:= TextXGlb;
  TextYLcl:= TextYGlb;
  SelectWorld(1);
  SelectWindow(1);
  CorrectMaxMinForWorld;
  Scale ;
```

```
        if AddBackgroundFlag
          then AddBackground;
        PlotPlanes;
        Beep;
        SelectWindow(3);
        TextXGlb:= TextXLcl;
        TextYGlb:= TextYLcl;
        Exiting('DrawOverheadView')
      end;
```

The variable "View" is set to Overhead, so the graphic utilities mentioned previously will work with the values of "ViewPts" for the overhead view.

The point data in "XfmdPts" are transformed to the coordinate system for the overhead view in the procedure OverheadViewTransform. The results of this transform are stored in the array "Vpts" for the overhead view. The points are transformed only for those parts which were moved or affected by a part which was moved. The X and Y values of these points are then given 3D perspective (Foley and Van Dam 1982).

The procedure FindHiddenLines determines the hidden surfaces for the overhead view using the algorithm described in the section titled "Theory of Hidden Surface Elimination." The indices of the planes which can be seen are stored in the array "SortPlane" along with the minimum-Z-values for each of these planes.

The procedure SortPlanes then sorts the data in the array "SortPlane" by the minimum-Z-values. The planes are sorted in decreasing order of minimum-Z-values.

The maximum and minimum values of X and Y for all the points in "Vpts" are determined in the procedure MaxMin.

The current X and Y position in window 3 are saved in local variables.

The graphics limits are selected by SelectWorld, and SelectWindow sets the current window to the overhead view window. The procedure CorrectMaxMinForWorld compares the maxima and minima data values against the limits for the window. If the limits in any direction (-X, -Y, +X, +Y) are greater than the corresponding max/min data values, the max/min data values are set to the limits value. Procedure Scale adjusts the max/min data values for the aspect ratio of the current graphics window and uses the adjusted values to determine the scaling factors for the view. The scaling factors are used to convert from view coordinates to the screen coordinates.

If a background has been created for the overhead view, the variable AddBackgroundFlag is true and the procedure AddBackground is called to plot the background.

The procedure PlotPlanes handles plotting the plane data to the overhead view window. Only the planes whose indices are stored in the array "SortPlane" are plotted. Each plane is filled as it is plotted to eliminate hidden surfaces.

Window 3 is selected by SelectWindow, and the previous X and Y positions in the window are restored. The procedure DrawOverheadView then returns to the routine ReadAndDraw.

The next procedure called by ReadAndDraw is the procedure to draw the shoulder view, DrawShoulderView. This routine is very similar to DrawOverheadView and uses many of the same graphic utility procedures.

When the robot moves forward or backward, then none of the values in the robot coordinate system are changed. The shoulder view of the robot does not change if none of the robot parts move. A Boolean variable named "DrawShoulderViewFlag" was created and will be true only if none of the robot parts are moved. The value of this variable is set by the command interpreter.

The listing of the procedure DrawShoulderView is as follows.

```
begin
  Entering('DrawShoulderView');
  if DrawShoulderViewFlag
    then
      begin
        View:= Shoulder;
        ShoulderViewTransform;
        FindHiddenLines;
        SortPlanes;
        MaxMin ;
        TextXLcl:= TextXGlb;
        TextYLcl:= TextYGlb;
        SelectWindow(2);
        Scale ;
        PlotPlanes;
        SelectWindow(3);
        TextXGlb:= TextXLcl;
        TextYGlb:= TextYLcl;
      end
```

```
        else DrawText(1,'NO CHANGE IN ',
                        'SHOULDER VIEW.',Yes);
      Beep;
      Exiting('DrawShoulderView')
    end;
```

If none of the robot parts have been moved, then the procedure DrawText will display the message "NO CHANGE IN THE SHOULDER VIEW" to window 3 and the routine will end, returning to ReadAndDraw. If a robot part has been moved, then the shoulder view is recalculated and redrawn.

The view is set to the value Shoulder. The data points in the array "XfmdPts" for the parts which have been moved are transformed by the shoulder transform matrix and stored in the array "Vpts."

The procedures FindHiddenLines, SortPlanes, and MaxMin perform the same function as in the DrawOverheadView procedure except that the data points "Vpts" for the shoulder view are used this time.

The current X and Y position within window 3 are stored in local variables, and the current window is set to the shoulder view window by calling SelectWindow.

Procedure Scale adjusts the max/min data values for the aspect ratio of the current graphics window and uses the adjusted values to determine the scaling factors for the view. The scaling factors are used to convert from view coordinates to the screen coordinates.

The procedure PlotPlanes handles plotting the plane data to the shoulder view window. Only the planes whose

indices are stored in the array "SortPlane" are plotted. Each plane is filled as it is plotted to eliminate hidden surfaces.

Window 3 is selected by SelectWindow, and the previous X and Y positions in the window are restored. The procedure DrawShoulderView then returns to the routine ReadAndDraw.

Overview of the Command Interpreter Software

The command interpreter software interprets high level language commands and sets variables according to the command. The command is either read from the source code file or is entered interactively by the operator. The command interpreter works with only one instruction at a time, and requires that one and only one instruction be entered on every line of the source code program.

The command interpreter software is controlled by one procedure in the graphics simulator, the procedure ReadSourceCode. The flow of ReadSourceCode is as follows.

```
begin
  Entering('ReadSourceCode');
  repeat
    GoodCommand:= false;
    GetSource;
    if not Done
      then Parser;
  until Done or Aborting or GoodCommand;
  Exiting('ReadSourceCode');
end;
```

In the procedure, a loop is started which continues until either a graphics simulator instruction has been successfully interpreted (GoodCommand is set true), the

operator has chosen to exit the program (Done is set true), or an abortive error has occurred (Aborting is set true). Within the loop, the variable GoodCommand is set to false and two procedures are executed. GetSource retrieves the next instruction to be executed or lets the operator choose to exit the program. If the operator did not choose to exit, the routine Parser is executed to determine what the command was and sets variables accordingly.

The next line in the source file was previously stored in a string variable named "NextCommand." Procedure GetSource displays the next command to the operator and prompts the operator to hit return to execute that command, hit escape to exit the program, or hit any other key to enter a command. If the operator hits escape, the variable Done is set true and the program eventually ends. If the operator enters a return, the next command is stored in the variable "CurrentCommand" and GetSource returns to ReadSourceCode. If the operator hits any other key, the graphics screen is stored in a memory array, a menu of the high level language instructions is displayed, and the operator is prompted to enter a command. When the operator enters the command and hits return, the command is stored in the variable "CurrentCommand" and the graphics screen is restored from the memory array to the display.

The parsing system for the graphics simulator was designed to be very similar to the compiler for the system.

This similarity will simplify the additions of new high level language commands to the overall system. One major exception to the similarity is that the parsing system for the compiler reads the source code directly from the source file while the parsing system for the graphics simulator reads the source code from the string variable "CurrentCommand." In spite of this difference, most of the procedures are very similar and even the same procedure names are used.

The parser scans the variable "CurrentCommand" character by character and breaks the code into tokens. Tokens are key words, parameters, delimeters, or scalars. The parser checks the tokens to determine whether they are part of the high level language and in the correct sequence. The parser consists of three procedures: Parser, Next_char, and Next_token.

Next_char reads the next character in the variable "CurrentCommand." All input characters are converted to upper-case letters which allows the high level language statements to be written in any combination of upper-case and lower-case letters.

The procedure Next_token assembles characters from the command into tokens. Whenever a blank space or semicolon is encountered, Next_token returns the completed token to the procedure from which it was called.

The procedure Parser controls the parsing of the source code. Parser calls the procedure Next_char to initialize the reading of the command. Each time a token is read from "CurrentCommand," Parser calls the procedure Controller which is the first procedure in the code generation phase of the interpreter.

The procedure Controller is called directly from Parser. The tokens which are returned to Controller from Next_token should be the first word of the high level command (keywords). If the token is "SPEAK" then procedure Speak_hero is called. If the token is "INITIALIZE," then the Initialize_hero procedure is called. If the token is "MEMORY," then the Memory_hero procedure is initiated, and if the token is "END," then the End_hero procedure is called. If the token is "GRAPHICS," then the procedure Graphics_hero is called. If the token is one of the robot parts, the procedure Move_hero is called. If the token is not a key word, an error is generated.

The "MEMORY" command is a compiler directive for setting the starting address of the program. The procedure Memory_hero displays a message to window 3 that the starting address has been set. The "SPEAK" command has the robot say a speech from its memory. The Speak_hero procedure also displays a message to window 3 that a robot is to say a speech. Since neither of these commands affect the position of the robot, the variable "GoodCommand" is set false, and

the loop within procedure ReadSource is repeated until a good command is entered.

The End_hero procedure is called only when an "END" statement has been entered. The flag Done is set true to cause the graphics simulator to end.

Initializing the hero robot moves all the robot parts back to their initial position. Procedure Initialize_hero sets the variable "Position" to its initial value for all the robot parts except the body. The variable "RecalculateFlag" is set true for every part, and the variable "DrawShoulderViewFlag" is set true. All the transform matrices will be recalculated and both views will be replotted.

The procedure Graphics_hero must read another token to determine the type of graphics command being issued. If the next token is valid, then one of three procedures will be called depending upon the command being issued: Graphics-Limits, GraphicsStartingpoint, or GraphicsBackground.

Procedure GraphicsLimits will set the world limits of the overhead view window to the values passed. This may cause a change in the overhead view. Since no part has moved, the variable "RecalculateFlag" is set false for every part, and "DrawShoulderViewFlag" is set false. The variable "GoodCommand" is set true. None of the parts will be transformed, but the overhead view will be redrawn.

Procedure GraphicsBackground will print a message that the graphics background statement has not yet been implemented. The variable "GoodCommand" is set false.

Procedure GraphicsStartingpoint will change the X and Y values of the variable "RobotPos" and change the position variable for the body part to the rotation angle from the command. The variable "RecalculateFlag" is set true for every part, but the variable "DrawShoulderViewFlag" is set false. All the transform matrices will be recalculated and only the overhead view will be replotted.

The procedure Move_hero is called when any robot part is moved. If the part name is "ARM" or "WRIST," another token is needed to determine the robot part. The variable "RbtPrt" is set to the value of the robot part moved.

After the robot part name has been determined, the token for the timing mode parameter is read. If the timing mode parameter is not "WAIT," an error is generated, but the parsing continues. Next, the position mode parameter token is parsed from the command, and the variable "AbsoluteM" is set true if the position mode is absolute or set false if the position mode is relative. A variable "Speed" is set to slow, medium, or fast depending upon the value of the speed token.

When all of the mode parameters have been determined, the procedure Amount_Validate is called. This procedure validates the values of move commands versus the limits of

the robot for each part of the robot. This routine handles both relative and absolute modes. If no errors occur (if the command is good), the position variables for the appropriate part are updated and the proper Recalculate flags are set for only those parts whose data points must be retransformed or for those parts affected by the moved part. The variables "GoodCommand" and "DrawShoulderViewFlag" are both set true. The limits of the robot for each part of the robot is shown in Table 3.

TABLE 3

LOWER AND UPPER LIMITS FOR ABSOLUTE AMOUNTS

ROBOT PART	LOWER LIMIT	UPPER LIMIT
Steering	-90.0	90.0
Body	-360.0	360.0
Head	0.0	350.0
Shoulder	0.0	150.0
Extend	0.0	5.0
Pivot	-90.0	90.0
Rotate	0.0	350.0
Gripper	0.0	3.5

This method of command parsing and interpretation allows for easy expansion of the high level language. As commands are added, new procedures to handle the command can be written and merged into the program.

GRAPHICS SIMULATOR USER MANUAL

The Robot Interactive Graphics Simulator interprets a text file containing instructions written in the Hero High Level Language. For each instruction the graphics simulator displays two views showing the position of the Hero Robot after the instruction has been executed. A status line indicating the current position of all the parts of the robot is also displayed.

A guide for operating the graphics simulator is in the section titled "Operating the Graphics Simulator." The section titled "Error Handling" discusses any errors which might occur while operating the graphics simulator and includes a complete description of the error messages displayed by the simulator.

The Robot Interactive Graphics Simulator is designed to be used on an IBM Personal Computer. An understanding of the IBM PC-DOS or MS-DOS operating system is assumed.

Operating the Graphics Simulator

Before Starting

The graphics simulator requires a source file, nine data files, and a font file. The graphics simulator will create a command log file and a token log file. The graphics simulator program itself is contained in a file

named RIGS.COM, and must be on the disk with the data files and the disk must be inserted into the logged drive of the computer.

The source file must be an ASCII text file which exists on a disk. Since commands may be input to the graphics simulator interactively, the source file file does not necessarily have to contain a High Level Language program, but the file must exist.

Nine data files are required for RIGS: MOVWHEEL.DTA, BODY.DTA, HEAD.DTA, ARM.DTA, EXTARM.DTA, WRPVT.DTA, WRROT.DTA, GRIPP.DTA, and GRIPM.DTA. The graphics simulator expects them all to be on directory \DATA. If an error occurs while opening or reading any of the data files, the program will abort.

The graphics simulator uses a graphics character font which is stored in the data file named "4x6.FON". The program expects to find the font file in the same directory as the program itself. If any error occurs while opening or reading the font file, the program will abort.

The graphics simulator will create a log file of all the commands which it attempts to execute and of all the error messages encountered. This file will be given the same filename as the source file but with the extension ".LOG". If an error occurs while opening the log file, the program will issue error message 32 and continue without logging any information.

The graphics simulator may also create a file into which it will write all of the tokens of the commands as they are interpreted. This file will be given the same filename as the source file but with the extension ".GTK". If an error occurs while opening the token file, the program will issue error message 27 and continue without logging any information. Note that this feature may be disabled by setting the value of variable "SaveTokens" to false in the initialization routine and recompiling the program. This feature may already be disabled on your disk.

Starting the Graphics Simulator

The graphics simulator can be started using the menu associated with the HICLASS system or on a stand-alone basis.

The HICLASS system menu can be started by typing the command HEROMENU from DOS. Once the menu is running, RIGS can be executed by typing G (for Graphics Simulator) followed by Y (for Yes).

To run graphics simulator stand-alone, simply type RIGS from the DOS prompt. The name of the source file can be passed to the program in the run-string, but this is optional. To start the graphics simulator and pass the source filename "SOURCE.HRO", the command

```
RIGS SOURCE.HRO
```

would be typed from DOS followed by a return.

The graphics simulator displays a welcome message in the middle of the screen, then displays the message

Reading Robot Part Data Files...please be patient.

If the program cannot open any of the part data files, an error message is displayed and the program aborts.

The graphics simulator then reads a character set from a font file ("4x6.FON"). If an error occurs while opening the font file, an error message is displayed and the program aborts.

After reading the part data files and font file, the program attempts to open the source file if a filename was passed in the run-string. If the source file cannot be opened, an error message is issued. If the filename was not passed in the run-string or that file could not be opened, the following prompt is displayed.

Enter the name of the source file:

Enter the complete filename of the source file, including the disk, directory path(s), filename, and extension. The operator is then asked to verify the filename by the prompt

Source file name is FILENAME.EXT.

Is this correct? (Y/n)

The response is Y or return for yes and N for no. If no is entered, the operator is prompted for the filename again. If an error occurs while opening the source file, an error message is displayed and the operator is again prompted to enter the filename. This process continues until the

operator enters a proper text filename or enters return (a blank filename). If a blank filename is entered, the following prompt is displayed.

You want to Abort the program.

Is this correct? (Y/n)

If you want to end the program, enter Y or return. If you want to continue, respond with N followed by a return and you will again be prompted for the filename.

The graphics simulator tries to open the command log file. The file will be given the same filename as the source file but with "LOG" as the extension. If the file cannot be opened, the program issues an error message and proceeds without logging commands.

The graphics simulator may try to open the token log file. The file will be given the same filename as the source file but with "GTK" as the extension. If the file cannot be opened, the program issues an error message and proceeds without logging the tokens. This optional file is typically used only when debugging the program, and may be eliminated from your version.

The graphics simulator now switches the screen mode to graphics and draws the status line, the overhead view, and the shoulder view of the robot. In this initial display, each of the robot parts are shown in the same position to which they would move following an INITIALIZE command.

Executing High Level Language Commands

High Level Language Commands may be entered into the graphics system from the source file or interactively. Both methods of entry are discussed within this section.

The graphics simulator will display at the bottom of the graphics screen the next line in the source file. If the end of the source file has been reached, the command END is displayed. For example, if the next line of the source file was MEMORY $0100; the prompt at the bottom of the screen would appear as follows.

```
THE NEXT SOURCE FILE LINE IS
MEMORY $0100;
HIT RETURN TO EXECUTE THE SOURCE LINE, ESCAPE TO EXIT,
OR ANY OTHER KEY TO TYPE IN A COMMAND.
```

If return is pressed, the command that was displayed is executed. If the escape key is pressed, the program ends immediately without giving you the opportunity of changing your mind. Pressing any other key on the keyboard will allow you to enter a command interactively.

If the option to enter a command interactively was chosen, the graphics screen is erased and a menu of all the available commands is displayed. The operator is prompted to enter a Hero Robot Language Command. Below this prompt is an inverse video line 64 characters long (the maximum length allowed for the command). As you type the command, it will be displayed in the inverse video line. To execute an interactive command, type the command and press return.

If you do not want to enter an interactive command, press return without typing any other characters first, and the program will prompt you again with the next source file line.

If the command executed is an END command, the graphics simulator will end without offering a chance for the operator to change his mind. If the command is a MEMORY command, the message

ROBOT ADDRESS SET TO LOCATION nnnn

is displayed with nnnn representing the address in the memory command. If the command is SPEAK, the message

SPEECH COMMAND #nn; NO MOVEMENT EXECUTED

is displayed with nn representing the speech number in the speech command. For both the SPEAK and MEMORY commands, no computations need to be performed on the parts data, so the operator is prompted to press any key to continue. After a key is pressed, the next source line is displayed as described at the beginning of this section.

If the command was not an END, MEMORY, or SPEAK command, the status line is updated with the new positions of the robot parts and the graphics simulator begins computing the new values of the parts data.

As the graphics simulator is performing the graphics data manipulation, several prompts will appear. Before the overhead view is drawn, the following prompts will be displayed in the bottom window.

```
TRANSFORMING COORDINATES.
TRANSFORMING FOR OVERHEAD VIEW.
DETERMINING HIDDEN LINES.
SORTING PLANE DATA.
DETERMINING MAXIMA AND MINIMA.
```

The overhead view is then drawn. If the command moved any of the robot parts, the following prompts are displayed in the bottom window before the shoulder view is drawn.

```
TRANSFORMING FOR SHOULDER VIEW.
DETERMINING HIDDEN LINES.
SORTING PLANE DATA.
DETERMINING MAXIMA AND MINIMA.
```

The shoulder view is then drawn. If the command did not move any of the robot parts, the following prompt is displayed.

```
NO CHANGE IN SHOULDER VIEW.
```

The next source file line is displayed as described at the beginning of this section.

Results of the Graphics Simulator

After running the graphics simulator program, a log file of the commands executed and the errors encountered in the run will exist unless the program could not open the file. This text file can be reviewed, edited, and renamed to become the next source file for another run of the graphics simulator.

Error Handling

The errors captured by the graphics simulator are file opening errors and errors occurring during command inter-

pretation. Only the file opening errors for the robot parts data files and the font file are errors which will cause the program to abort. For all other errors, an error message is displayed and the operator must acknowledge the error by pressing the escape key before the program will continue.

File errors are defined by the error number in the error messages section. The operator must determine the final cause of the file problem and fix the problem.

Command interpretation errors can be fixed by changing the source code.

The error messages for RIGS are described in detail in the following section.

Error Messages

There are a total of 25 errors which are identified by the Robot Interactive Graphics Simulator. These messages are displayed in white characters on a red background and are also written to the log file (unless the log file could not be opened). This section provides a more complete description of the error which occurred, and may suggest how to correct the error condition.

If you find typing a problem, please note that the graphics simulator requires only the first character for almost every command. Only the "GRAPHICS" and "GRIPPER" commands require three characters to differentiate them from each other.

Error 1: Expecting Keyword. This error indicates that the first token in a high level language statement was not a keyword. Acceptable keywords are INITIALIZE, END, GRAPHICS, MEMORY, SPEAK, ARM, DRIVE, GRIPPER, HEAD, TURN, and WRIST. Check the spelling of the first word in the command.

Error 2: Expecting Continue or Wait for Speak Command. The word following a "SPEAK" statement was neither "CONTINUE" nor "WAIT", the only acceptable timing parameters. Insert the appropriate parameter in the command or correct the spelling of the current parameter.

Error 3: Invalid Speech Number. The number after the "SPEAK" command is out of range. The number must be an integer value greater than or equal to 1 and less than or equal to 19.

Error 4: Expecting Semicolon. This error means that a non-motor move command was not followed by the required semicolon. The statements which require semicolons are "END," "INITIALIZE," "MEMORY," "GRAPHICS," and "SPEAK." If this error occurs, add a semicolon to the end of the appropriate statement(s).

Error 5: Invalid Graphics Parameter. The next word after a "GRAPHICS" command was not a valid choice. The acceptable parameters are "BACKGROUND," "LIMITS," or "STARTINGPOINT."

Error 6: Invalid Wrist Command--Expecting Pivot or Rotate. The second token in the command was not "PIVOT" or "ROTATE" or an acceptable abbreviation of these parameters. This parameter is needed to distinguish between the two wrist motors.

Error 7: Invalid Arm Command--Expecting Extend or Shoulder. The second token in the command was not "SHOULDER" or "EXTEND" or an acceptable abbreviation of these parameters. This parameter is needed to distinguish between the two arm motors.

Error 8: Expecting Continue or Wait for Move Command. The second token of the command is not "CONTINUE" or "WAIT" or an acceptable abbreviation of these parameters. Insert the appropriate parameter in the command statement or correct the spelling of the current parameter. Note that although continue mode is valid for the HEROIC compiler, the graphics simulator does not support this mode, so the timing mode parameter must be "WAIT."

Error 9: Expecting Absolute or Relative. The third token in a move command was not "ABSOLUTE" or "RELATIVE" or an acceptable abbreviation of these parameters. The absolute/relative parameter is not used for DRIVE commands.

Error 10: Expecting Motor Speed. The fourth token in the command line was not "SLOW," "MEDIUM," "FAST," or an

acceptable abbreviation of those words. If this error occurs, add the desired speed or correct the spelling of the speed parameter.

Error 11: Invalid Motor Command. This error is generated if the first token is not one of the robot's motors or a valid abbreviation of a motor name.

Error 12: Invalid Absolute Motor Move Value. This error occurs when the amount parameter specified in the absolute move command is not a scalar, or if the amount specified is less than the part's lower limit or greater than the part's upper limit. Check Table 3 to verify that the amount specified is valid.

Error 13: Invalid Relative Motor Move Value. This error occurs when the relative amount specified is not a scalar, or if the amount specified would move the part beyond its upper or lower limits.

Error 21: Could not open one of the robot data files. Nine data files are required for RIGS: MOVWHEEL.DTA, BODY.DTA, HEAD.DTA, ARM.DTA, EXTARM.DTA, WRPVT.DTA, WRROT.DTA, GRIPP.DTA, and GRIPM.DTA. The graphics simulator expects them all to be on directory \DATA. Either one of the files is missing or the file is not in the proper subdirectory.

Error 22: Could not open the font file (4x6.FON). The graphics simulator uses a graphics character font which is stored in the data file named "4x6.FON". The program expects to find the font file in the same directory as the program. Either this file is missing or is not in the proper directory.

Error 23: Could not open the source file FILENAME. The file name which was either passed in the run-string of the program or input by the operator could not be opened. The source file's subdirectory must be included if the source file is not in the current directory. The operator will be prompted to re-enter the source file name. If the operator wants to exit the program, entering only a return will begin a program ending sequence. Note that a valid text file name MUST be entered for the source file name.

Error 24: Text string set to DrawText contains illegal characters. The routine which prints messages in window 3 of the graphics screen was sent a character which is not in its character set. This error message was included to aid in debugging the program and should not occur during operation.

Error 25: Please respond with Y or N or hit Return for the default. The operator has been asked a question for which the only valid responses are Y for yes, N for no, or a

return for the default answer. Press the escape key to be asked the same question again.

Error 26: Robot moved while Steering Wheel was turned; NOT SUPPORTED. The graphics simulator does not support moving the body of the robot with a DRIVE statement while the Steering Wheel is at any position other that zero degrees absolute. When escape is pressed, the program will move the robot as if the wheel was set to zero degrees. This is a limitation of the graphics simulator.

Error 27: Could not open tokens file FILENAME.GTK; tokens not saved. The program cannot open the file in which it wanted to save the command tokens. The file would have been given the same name as the source file but with the extension GTK. The graphics simulator will not save the tokens. It is possible that your disk is full.

Error 28: Invalid limits specifications. The command "GRAPHICS LIMITS" requires four real-number parameters: a LowX position, a LowY position, a HighX position, and a HighY position. Each of these parameters should have the units of feet. One of the parameters was not a real number. Note that real numbers less than one must start with a zero followed by a decimal point.

Error 29: Invalid starting point specifications. The command "GRAPHICS STARTINGPOINT" requires three real-number

parameters: an X position (feet), a Y position (feet), and an angle (degrees). One of the parameters was not a real number. Note that real numbers less than one must start with a zero followed by a decimal point.

Error 30: Invalid number of parameters. A "GRAPHICS" command was passed the incorrect number of parameters. A "STARTINGPOINT" command requires 3 parameters, while a "LIMITS" command requires 4 parameters.

Error 31: CONT is NOT SUPPORTED; command treated as WAIT. The Robot Interactive Graphics Simulator does not support the "CONTINUE" timing mode parameter. The graphics simulator will proceed as if the parameter had been "WAIT," but the source file should be changed to avoid future errors.

Error 32: Could not open log file FILENAME.LOG; commands not logged. The program tried unsuccessfully to open a file into which it would write the commands executed and the errors encountered. The program will continue without logging commands. Your disk or the current directory may be full.

CONCLUSIONS

The current robotic training devices, such as the Hero, do not have programming systems that are like the industrial robots, and as such are limited in their effectiveness as a training tool.

High level languages and graphics simulators have been developed for most industrial robots to improve programming efficiency. A high level language and graphics simulator system was developed for the Hero Robot to improve its effectiveness as an education tool and to provide the user with the concept of the tools available on industrial systems.

The language developed is easy to use and its English command structure is easy to learn. The consistent parameter structure also reduces the time required to learn the language. The ability to control all of the motors of the robot has been included in the language. The language also has the ability to reference preprogrammed speeches. The language is structured to allow for future expansions.

The Robot Interactive Graphics Simulator lets the operator visually verify the movements made by the Hero Robot in response to the program or interactive commands. The graphics simulator interprets high level language commands and displays the correct position of the robot in

response to those commands. Two views of the robot and a position status line are displayed to provide the operator with complete information about the current position of the robot. The views of the robot are shown in color graphics with the hidden lines removed.

Both the Hero High Level Language and the Robot Interactive Graphics Simulator meet or exceed all the requirements set forth at the outset of the development process.

RECOMMENDATIONS

In order to prevent a serious shortage of trained robot operators and programmers, more emphasis should be placed on the development of robotic training systems.

The rapid growth in industrial robotic applications is largely due to the improvements in robot sensors and vision systems. The sensory capabilities of the Hero Robot should be added to the high level language to enable operators to become more familiar with the operation of modern industrial robots with sensing systems.

Future research should be directed toward providing the HICLASS system with a coordinate system that would be consistent throughout the language, compiler, and graphics simulator. The graphics simulator uses a coordinate system for moving the robot relative to the background, but the coordinate system is not used by the language or the compiler. This would enable the development of a task-based programming language system. Such a system would accept statements such as "move object to point A." Task-oriented languages are being developed for industrial robots, and the development of such a system for the Hero would be the next step in providing a training system which matches the robots used in industry.

The Robot Interactive Graphics System should be expanded to allow for CONTINUE mode statements. This would require the inclusion of time as a parameter, changing the system to a dynamic simulation system.

Although most industrial robots are not completely mobile today, this will become more common as technological advances are made. The graphics simulator should be expanded to include simulation of the robot when moving with the steering wheel turned to a position other than straight ahead. Since the steering wheel can be turned while the robot is driving forward or back, the addition of time as a parameter, mentioned in the previous paragraph, would preclude this addition.

APPENDIX A

SAMPLE OUTPUT OF GRAPHICS SIMULATOR

The following pages contain a High Level Language program which was executed on the graphics simulator. The source program used by the graphics simulator is listed. Copies of the graphics screen are included.

Sample Source Program

```
LINE NUMBER     HIGH LEVEL LANGUAGE COMMAND

     1          MEMORY $0100;
     2          GRAPHICS LIMITS -2 -2 2 2;
     3          GRAPHICS STARTINGPOINT -1 -1 -45;
     4          INITIALIZE;
     5          ARM SHOULDER WAIT ABS FAST 90
     6          WRIST ROTATE WAIT REL MED 90
     7          GRIPPER WAIT ABS SLOW 3.5
     8          ARM EXTEND WAIT ABS FAST 5.0
     9          HEAD WAIT REL FAST 90
     10         DRIVE WAIT FAST 2.0
     11         SPEAK WAIT 1;
     12         TURN WAIT REL FAST -25.0
     13         END;
```

Sample Graphics Screen Copies

Figures 2 through 14 are copies of the graphics screen of the IBM Personal Computer after each command was of the sample program was executed.

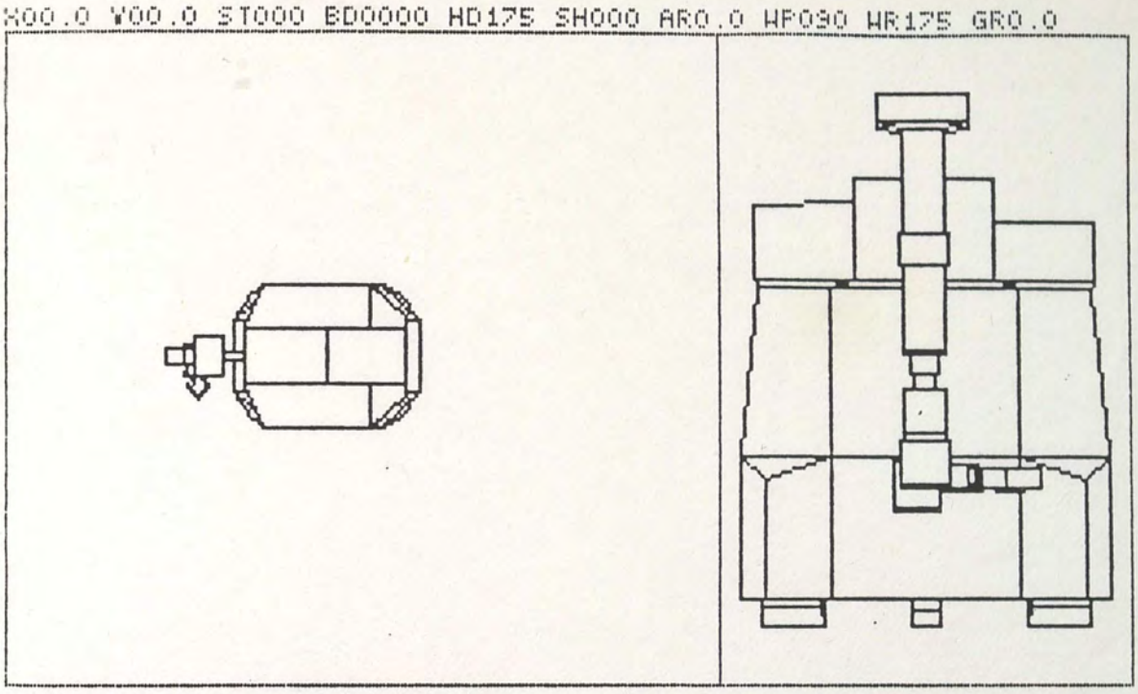

Figure 2. Initial Graphics Simulator Screen

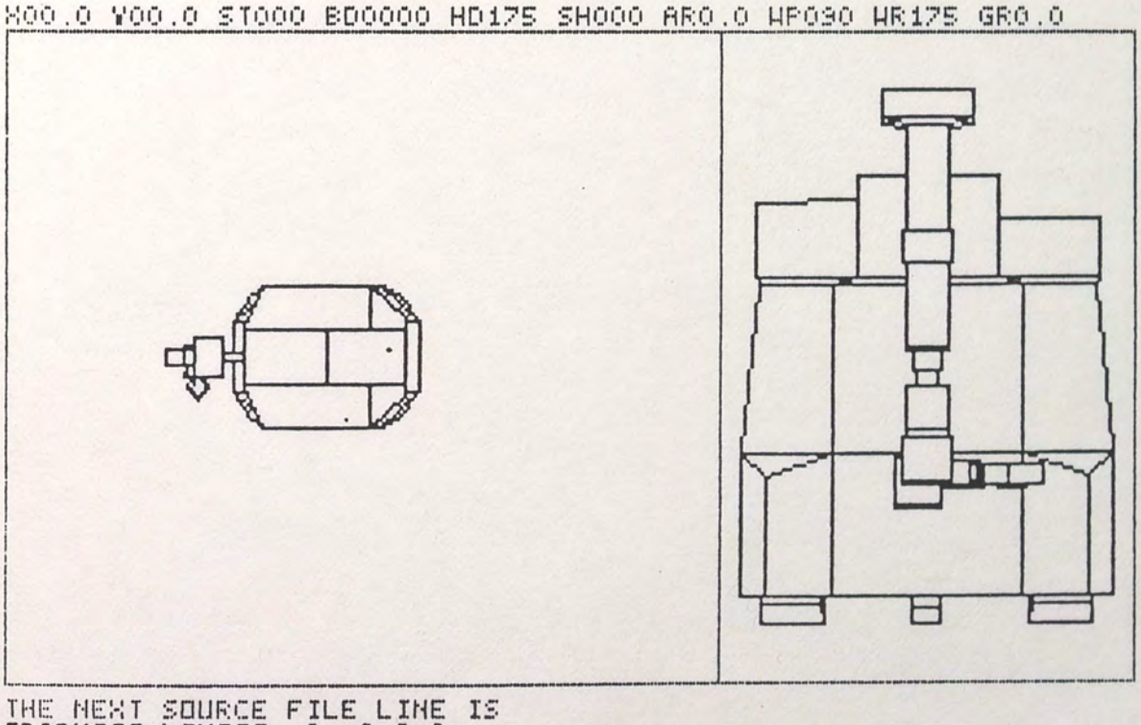

Figure 3. Screen After Memory Command

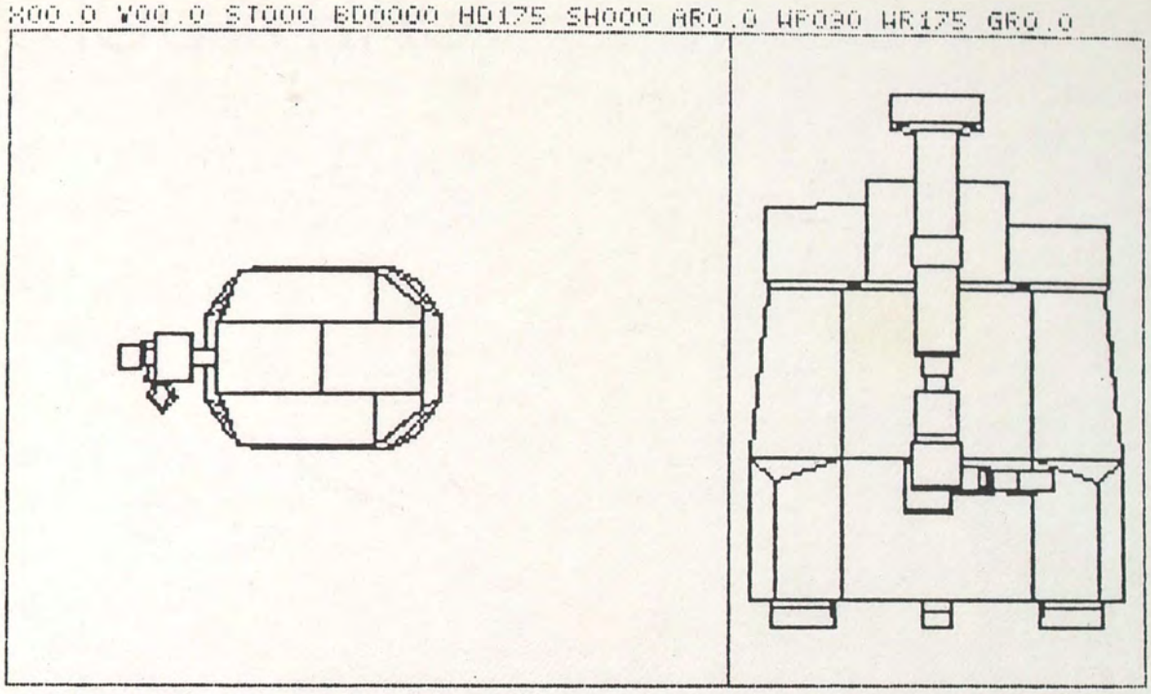

Figure 4. Screen After Graphics Limits Command

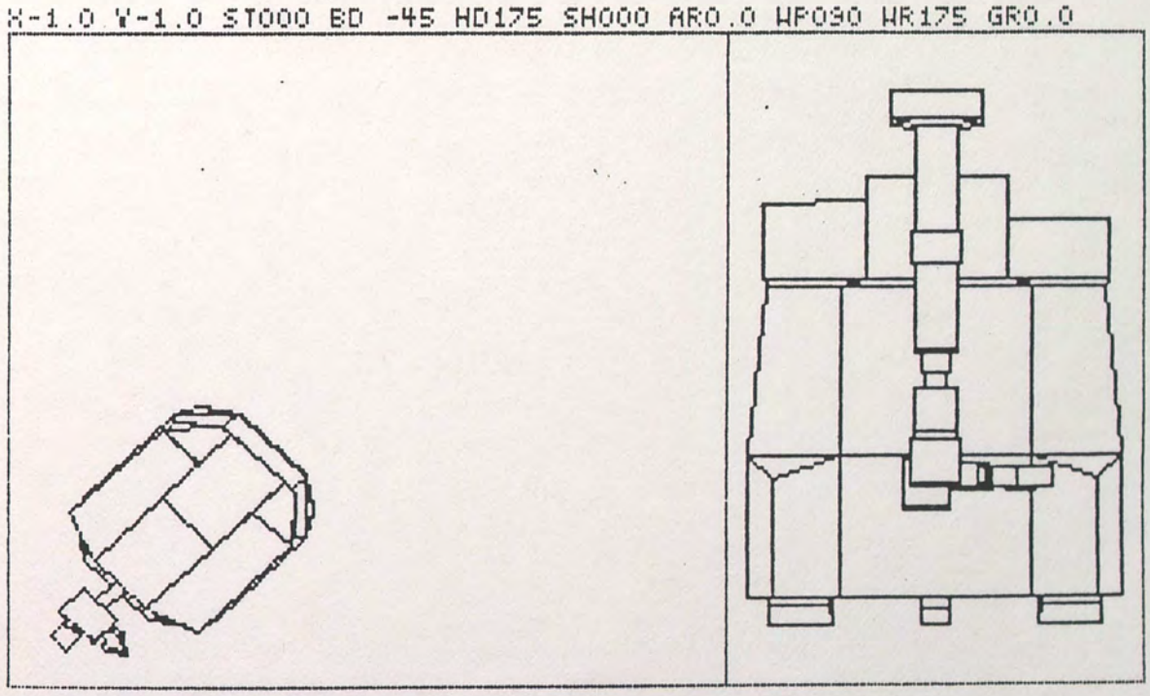

Figure 5. Screen After Graphics Startingpoint Command

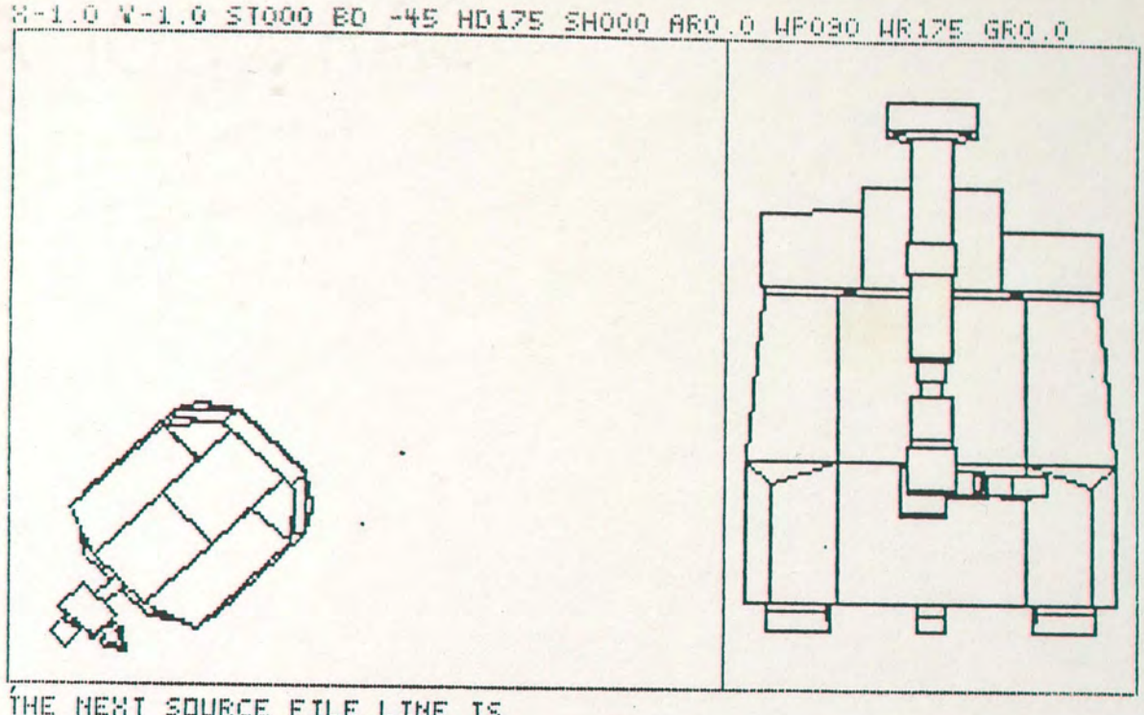

Figure 6. Screen After Initialize Command

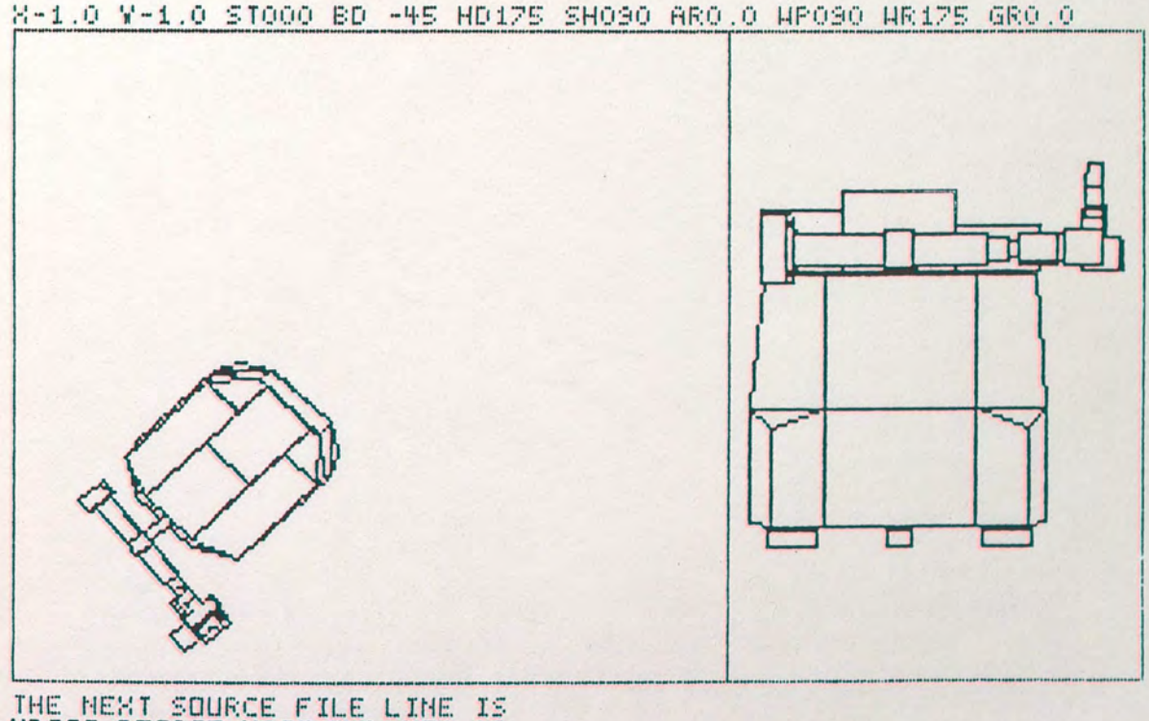

Figure 7. Screen After Arm Shoulder Command

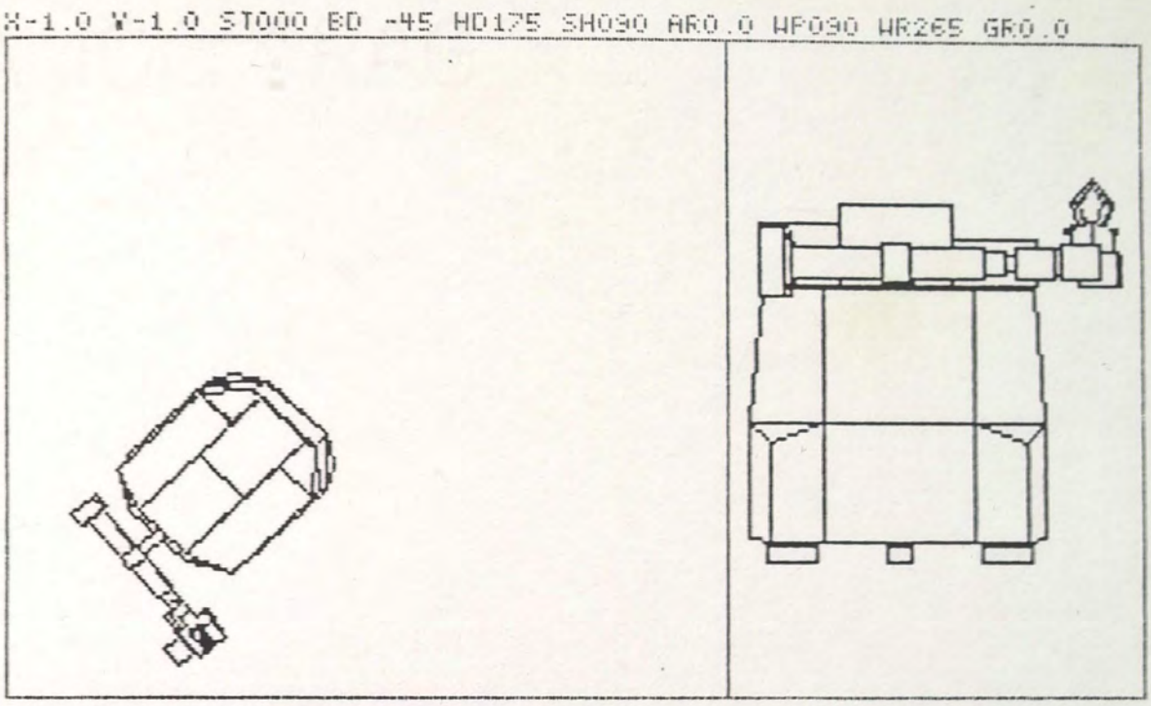

Figure 8. Screen After Wrist Rotate Command

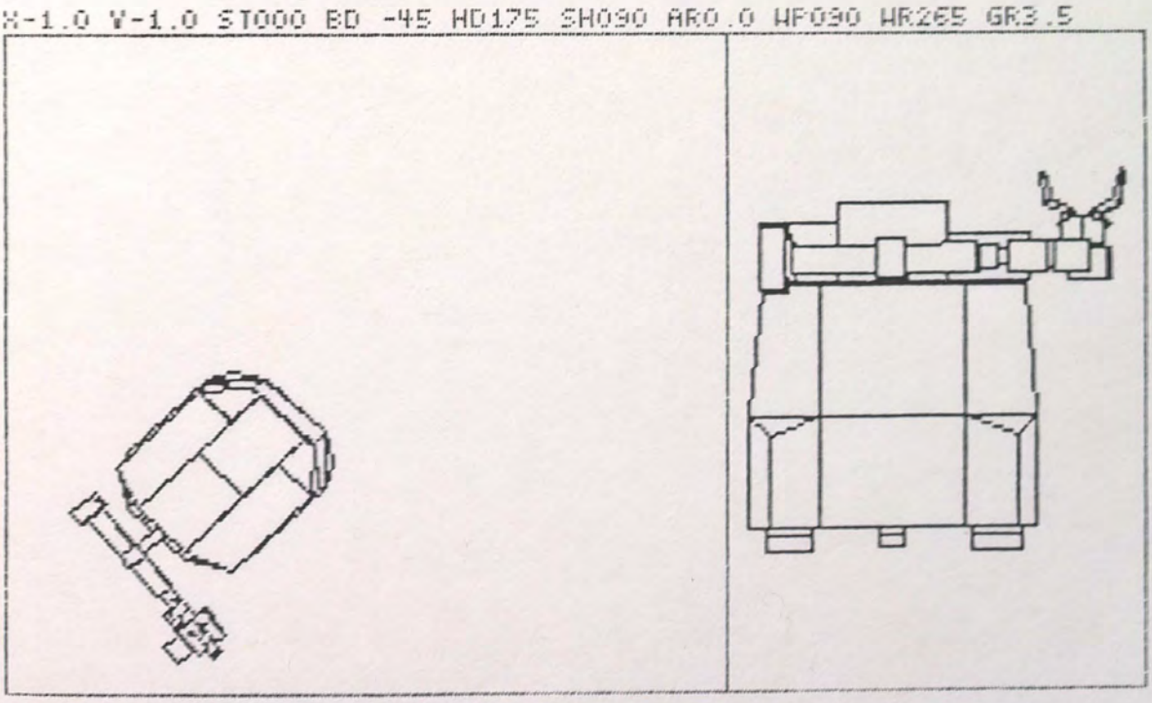

Figure 9. Screen After Gripper Command

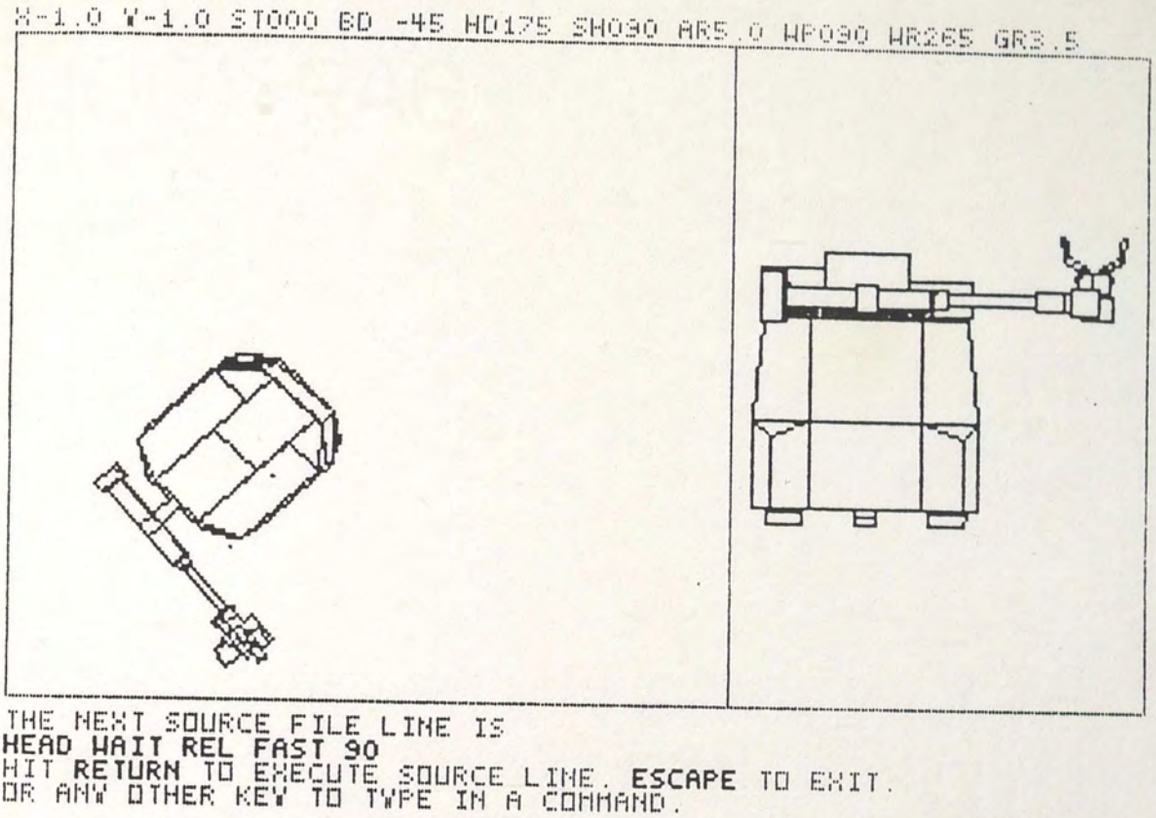

Figure 10. Screen After Arm Extend Command

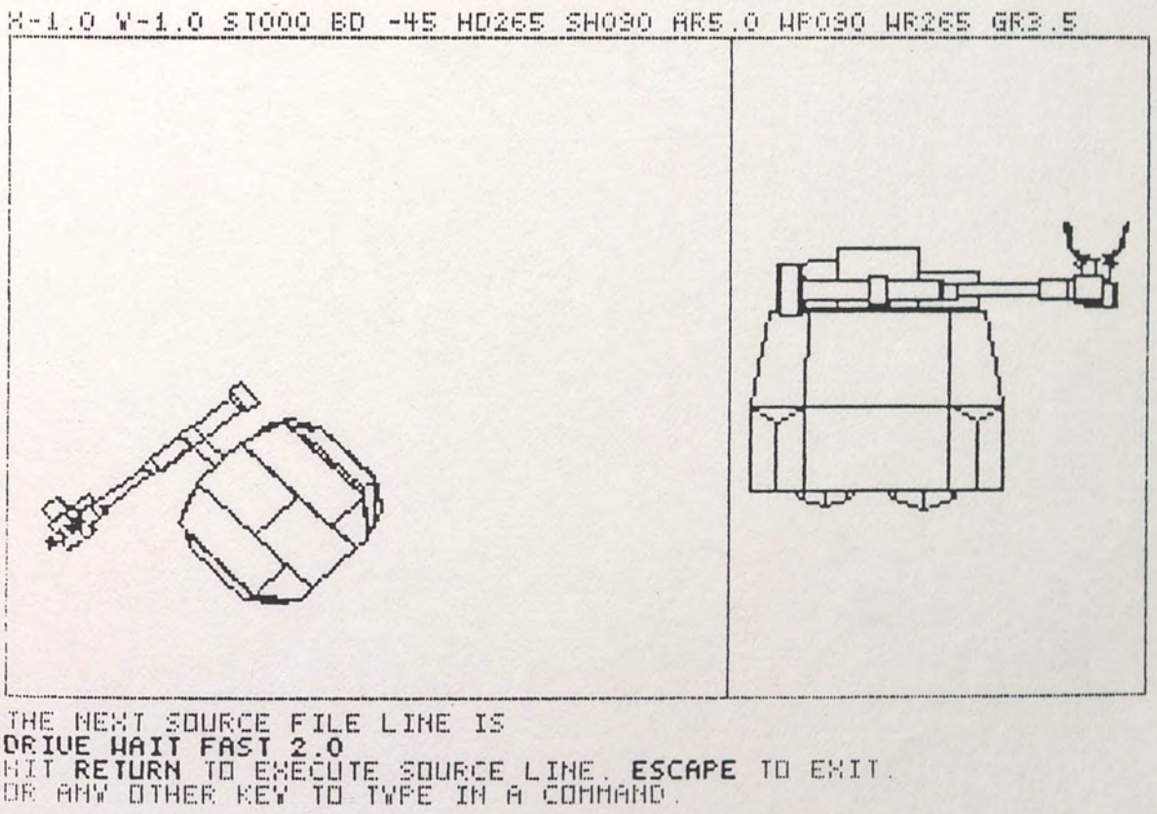

Figure 11. Screen After Head Command

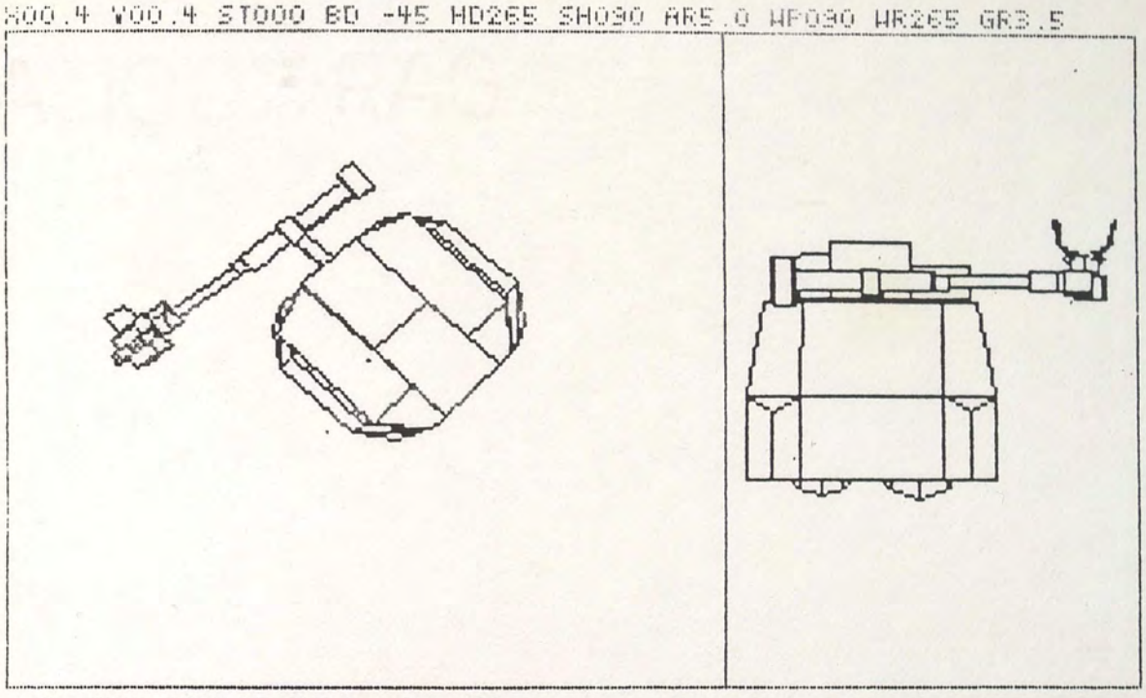

Figure 12. Screen After Drive Command

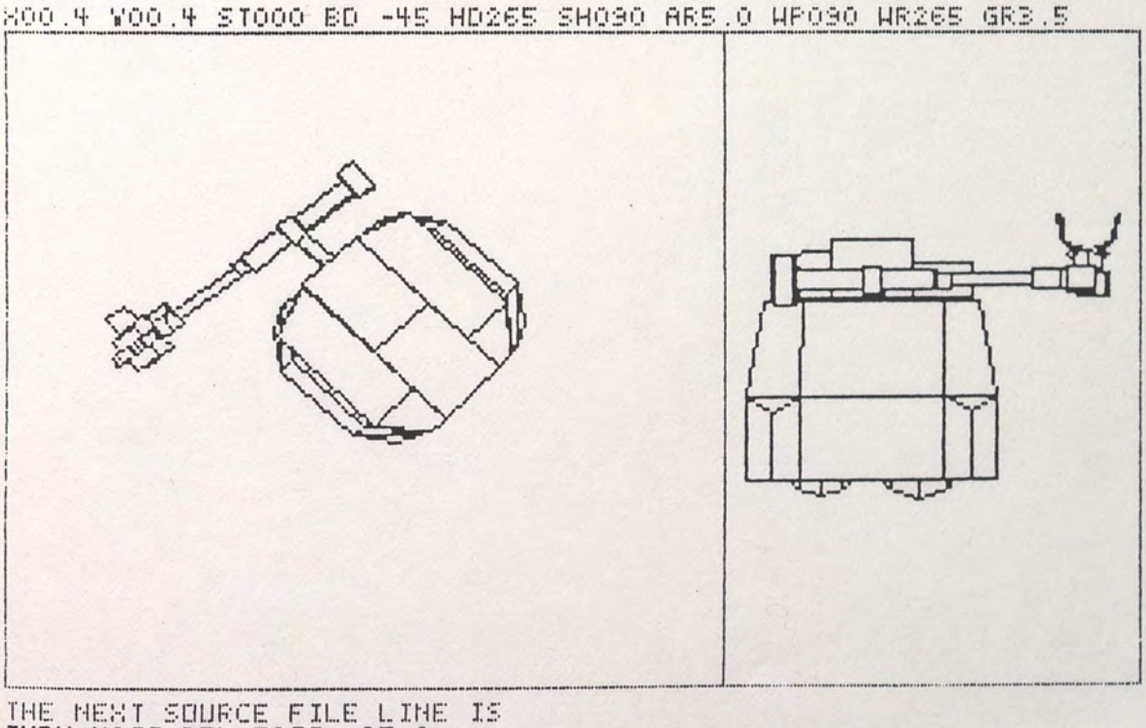

Figure 13. Screen After Speak Command

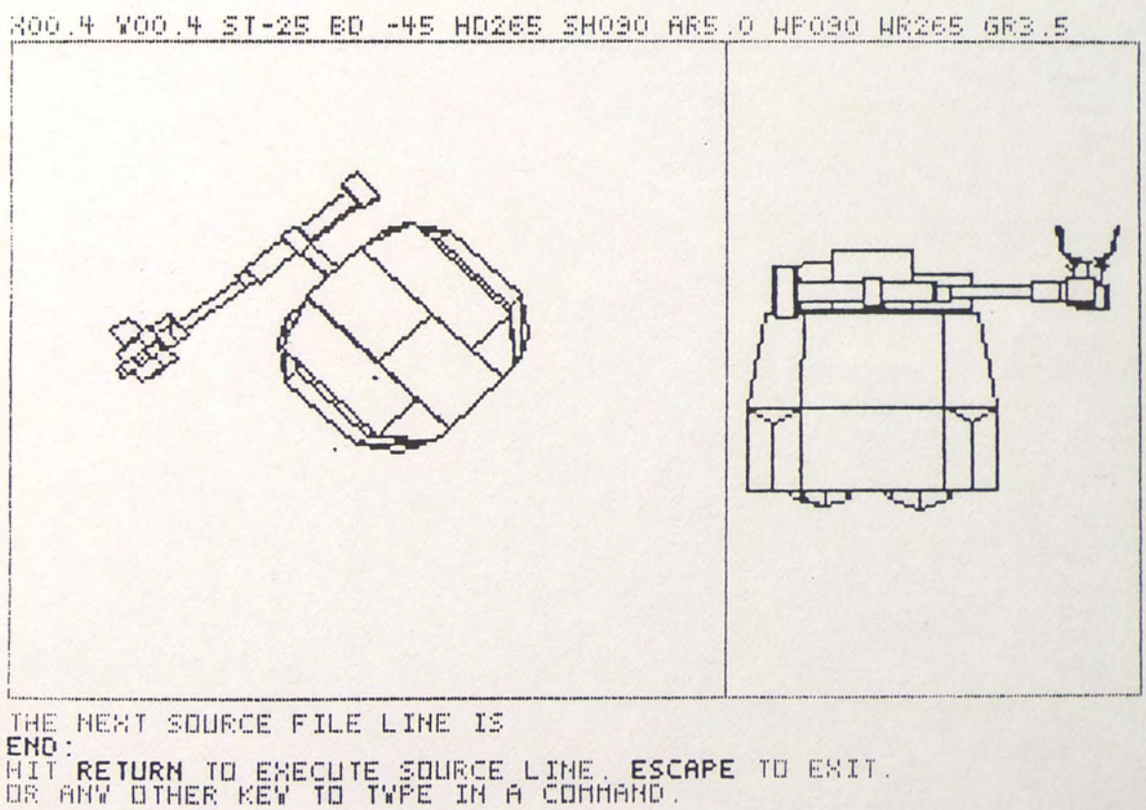

Figure 14. Screen After Turn Command

APPENDIX B

GRAPHICS SIMULATOR PROGRAM LISTING

The entire Robot Interactive Graphics Simulator program is listed in the following pages.

```
{ ***************************************************
  *                    R. I. G. S.                  *
  *      ( Robot Interactive Graphics Simulator )   *
  *                    Version 3.0                  *
  *               by Kenneth J. Sizemore            *
  *************************************************** }

program Rigs(Input,Output);
{
This program will interpret commands written in the Hero Robot High Level
Language and draw an overhead view and arm view of a Hero Robot after each
command is executed.  Commands are read in from a source file and may also
be typed in interactively by the operator.  This program can be used to
visually verify the movements of the robot before the program is actually
executed by the robot.

The name of the source file may be included in the run string of the program.
If the source file is not included in the run string, the program will prompt
the operator for the file name.  The program will create a tokens file and a
log file.  The tokens file will have the same name as the source file with
the extension GIK.  The log file will have the same name as the source file
but with the extension LOG.


}
```

```
                                                                    {.PA}
{ GLOBAL CONSTANTS }

const
  Esc: char = #27;                                  { Standard Characters }
                  { Screen Information and Screen Related Program Parameters }
  XMaxGlb = 319;
  YMaxGlb = 199;
  ScreenRows: integer = 25;
  ScreenCols: integer = 80;
  FontStrLen = 64;
  Yes : boolean = true ;       { Utility Parameters - used in YesNo function }
  No : boolean = false ;
  WidthArray = 4 ;               { Width of all point and transform matrices }
  MaxWorldsGlb = 4;
  MaxWindowsGlb = 4;
  MaxNumBgLines = 200;
  ScreenDistance: real = 30.0;     { Distance in inches of robot from viewer }

                                                                   {.CP18}
{ GLOBAL TYPES }

type
  AnyString = string[80];                  { Types used in utility routines }
  CharSet   = set of char;
  String3   = string[3];
  MaxPoints  = 1..300 ;                          { Drawing Data Array Ranges }
  MaxLines   = 1..600 ;
  MaxPlanes  = 1..300 ;
  MaxBgLines = 1..MaxNumBgLines ;
  ArrayWidth = 1..WidthArray ;
  Edge       = array[ArrayWidth] of integer ;          { Drawing Data Types }
  Vertex     = array[ArrayWidth] of integer ;
  EndPoint   = array[1..2] of integer ;
  Coordinate = array[ArrayWidth] of real ;
  PointType = array[MaxPoints] of Coordinate;
  ViewType = (Overhead, Shoulder);
  ViewRec = record                                                  {.CP3}
              Vpts: PointType;
            end;
  Line = record                                                     {.CP3}
           EndPts: EndPoint ;
         end;
  Plane = record                                                    {.CP5}
              Edges: Edge;
              Vertices: Vertex;
              PFill: boolean;
            end;
```

```
SortRec = record                                                    {.CP5}
            PlaneIndex: integer;
            MinZ: real;
            FillFlag: boolean;
          end;
RobotParts = (Steering,Body,Head,Arm,ExtendArm,WrPvt,WrRot,Gripp,Gripm);
IndexRec = record                                                   {.CP4}
            First,
            Last: integer;
          end;
TransformMatrix = array[ArrayWidth] of Coordinate;
RobotRec = record                                                   {.CP7}
            Index: array[1..3] of IndexRec;
            Position: real;
            Absolutes: array[1..2] of real;
            RecalculateFlag: boolean;
            T: TransformMatrix;
          end;
BgLineRec = record                          { BackgroundLine Types } {.CP3}
             BgXY: Coordinate;
           end;
character = array[1..3] of byte;                  { Font Types }    {.CP3}
CharArray = array[32..126] of character;
FontString = string[FontStrLen];
GraphicsScreenHeap = ^GraphicsScreenRec;      { Screen Save Types } {.CP4}
GraphicsScreenRec = record
                      GraphicsScreenBuffer: array[0..32006] of byte;
                    end;
WorldType = record                              { Windowing Types } {.CP9}
             x1,y1,x2,y2:real;
            end;
WindowType = record
              x1,y1,x2,y2:integer;
             end;
worlds = array [1..MaxWorldsGlb] of WorldType;
windows = array [1..MaxWindowsGlb] of WindowType;
New_Token = string[FontStrLen];
RobotSpeed = (slow, med, fast);                  { Robot Command Type }
```

```
{ GLOBAL VARIABLES }                                                    {.PA}

var                                                          { Global flags }
  Aborting, Debug, Debug_Flow, Done, Echo, GraphicsOn, LogCommands,
    NewAngles, NewRobotPos, SaveTokens, SourceOpen, TokenOpen,
    VaryDistance: boolean ;
  RecordNumber: integer;                                  { File Related data }
  Source, BgFile, LogFile, TokFile: text;
  FileName: string[80];
  CharFont: CharArray;
  Points, XfmdPts: PointType;                                { Figure Data }
  ViewPts: array[ViewType] of ViewRec;
  View: ViewType;
  Tfm : TransformMatrix;
  Lines : array[MaxLines] of Line ;
  Planes: array[MaxPlanes] of Plane;
  Parts: array[RobotParts] of RobotRec;
  RobotPos: Coordinate;
  BodyIncrement: real;
  SortPlane: array[MaxPlanes] of SortRec;
  NumBgLines, NumPoints, NumLines, NumPlanes, NumSortPlanes : integer ;
  CenterCol, ErrorLineNum: integer;
  Xmin, Xmax, Ymin, Ymax : real ;                        { Screen Scaling Data }
  Xslope, Xint, Yslope, Yint : real ;
  GraphicsScreenPtr: GraphicsScreenHeap;                  { Screen Saving Data }
  AddBackgroundFlag: boolean;                                { Windowing Data }
  TempBgLine: BgLineRec;
  BgLine: array[MaxBgLines] of BgLineRec;
  BgMaxMin: Coordinate;
  X1WldGlb, X2WldGlb, Y1WldGlb, Y2WldGlb, AxGlb, AyGlb, BxGlb, ByGlb: real;
  X1RefGlb, X2RefGlb, Y1RefGlb, Y2RefGlb: integer;
  TextXGlb, TextYGlb, TextXLcl, TextYLcl: integer;
  MaxWorldGlb, MaxWindowGlb, WindowNdxGlb: integer;
  X1Glb, X2Glb, Y1Glb, Y2Glb: integer;
  world: worlds;
  Gwindow: windows;
  ErrorCount, TokenCount, Result: integer;              { Parsing Data }
  CommandPosition, CommandLength: integer;
  Com, CurrentCommand, NextCommand, Semi_Token, TempCom, Token: New_Token;
  Continue, EndOfCommandLine, GoodCommand, Keyword, Move, Relative: boolean;
  Charac: char;
  Token_Val: real;
  Speed: RobotSpeed;

{$I GRAPH.P  }
```

```
{-------------------------------------------------------------------------
**** UTILITY PROCEDURES & FUNCTIONS ****
-------------------------------------------------------------------------}

function Upc(S: AnyString): AnyString;
{
  PURPOSE: Converts the string passed to all uppercase characters.
  INPUTS: S: AnyString ( any string of up to 255 characters)
  OUTPUTS: returns the same string with all characters converted to uppercase.
}
  var
    I: integer;
  begin
    for I:=1 to Length(S) do S[I]:=UpCase(S[I]);
    Upc := S
  end;

{-------------------------------------------------------------------------}
                                                                {.CP14}

procedure Beep ;
{
  PURPOSE: outputs an audible tone to the operator (i.e., it beeps).
  OUTPUTS: a beep for a tenth of a second.
}
  begin ;
    sound(550) ;  { A nice, low beep }
    delay(50) ;  { Nowhere near as long as printing a control-G }
    nosound ;
  end ;

{-------------------------------------------------------------------------}
```

{.CP32}

```
function OpenText(var TempFil: text; TempName: AnyString; Rw:boolean): boolean;
{
  PURPOSE: This function Opens the text file of the name passed as a
           parameter.  The file is opened for reading.
  INPUTS:
    TempFil: The text file variable.
    TempName: The file name
    Rw: Read/WriteNot flag.  Set true if the file should be opened for
        reading, false if the file should be opened for writing.
  OUTPUTS: true if the file opened successfully, otherwise false.
}
  var
    TempFlag: boolean;
  begin
    assign(TempFil, TempName);
    if Rw
      then
        begin
          {$I-} Reset(TempFil) {$I+};
          TempFlag:= (IOresult = 0)
        end
      else
        begin
          {$I-} Rewrite(TempFil) {$I+};
          TempFlag:= (IOresult = 0)
        end;
    OpenText:= TempFlag
  end;

{------------------------------------------------------------------------------}
```

{.CP11}

```
function DegToRad(Angle: real):real;
{
PURPOSE:  This function converts angles from degrees to radians.  Radians
are the units used for all trig functions within Turbo.
}
  begin
    DegToRad:= pi * (Angle / 180.0 )
  end;

{------------------------------------------------------------------------------}
```

```
                                                                    {.CP11}
procedure Entering(Name: AnyString);
{
PURPOSE: This routine is used to help debug program flow.
}
  begin
    if Debug_flow then writeln(lst,' entering ',Upc(Name),'.')
  end;

{------------------------------------------------------------------------}
                                                                    {.CP11}
procedure Exiting(Name: AnyString);
{
PURPOSE: This routine is used to help debug program flow.
}
  begin
    if Debug_flow then writeln(lst,' exiting  ',Upc(Name),'.')
  end;

{------------------------------------------------------------------------}
                                                                    {.CP22}
procedure WaitForEsc;
{
PURPOSE: This procedure keeps updating the time until the operator presses
         the escape key.  If the operator presses any other key, the
         routine calls Beep.
}
  var
    EscPressed: boolean;
    ch: char;
  begin
    repeat
      repeat
      until KeyPressed;
      read(kbd,ch);
      EscPressed:= (ch = Esc) and (not KeyPressed);
      if not EscPressed then Beep;
    until EscPressed;
  end;

{------------------------------------------------------------------------}
```

```
                                                                    {.CP16}
procedure DefineWorld(i:integer;
                      X_1,Y_1,X_2,Y_2:real);
  begin
    with world[i] do
      begin
        x1:=X_1;
        y1:=Y_1;
        x2:=X_2;
        y2:=Y_2;
        if i>MaxWorldGlb
          then MaxWorldGlb:=i;
      end
  end;

{-------------------------------------------------------------------------}
                                                                    {.CP13}
procedure SelectWorld(i:integer);
  begin
    with world[i] do
      begin
        X1WldGlb:=x1;
        Y1WldGlb:=y1;
        X2WldGlb:=x2;
        Y2WldGlb:=y2;
      end
  end;

{-------------------------------------------------------------------------}
                                                                    {.CP14}
procedure DefineWindow(i,X_1,Y_1,X_2,Y_2:integer);
  begin
    with Gwindow[i] do
      begin
        x1:=X_1;
        y1:=Y_1;
        x2:=X_2;
        y2:=Y_2;
        if i>MaxWindowGlb then MaxWindowGlb:=i;
      end;
  end;

{-------------------------------------------------------------------------}
```

```
                                                                    {.CP18}
procedure DrawBorder(i:integer);
  begin
    with Gwindow[i] do
      begin
        WindowNdxGlb:=i;
        GraphWindow(0,0,XMaxGlb,YMaxGlb);
        {
        Draw a border around the window
        }
        Draw(x1-1, y1-1, x2+1, y1-1, 1);
        Draw(x2+1, y1-1, x2+1, y2+1, 1);
        Draw(x2+1, y2+1, x1-1, y2+1, 1);
        Draw(x1-1, y2+1, x1-1, y1-1, 1);
      end
  end;

{-------------------------------------------------------------------------}
                                                                    {.CP17}
procedure SelectWindow(i:integer);
  begin
    with Gwindow[i] do
      begin
        WindowNdxGlb:=i;
        GraphWindow(x1,y1,x2,y2);
        X1RefGlb:=0;
        Y1RefGlb:=0;
        X2RefGlb:=x2-x1;
        Y2RefGlb:=y2-y1;
      end;
    TextXGlb:= 0;
    TextYGlb:= 5;
  end;

{-------------------------------------------------------------------------}
```

```
                                                                    {.CP18}
procedure InitGraphic;
  var
      i:integer;

  begin
    for i:=1 to MaxWorldsGlb do
      DefineWorld(i,0,0,XMaxGlb,YMaxGlb);
    MaxWorldGlb:=1;
    for i:=1 to MaxWindowsGlb do
      DefineWindow(i,0,0,XMaxGlb,YMaxGlb);
    MaxWindowGlb:=1;
    WindowNdxGlb:=1;
    SelectWorld(1);
    SelectWindow(1);
  end;

{-------------------------------------------------------------------------}
                                                                    {.CP19}

procedure SaveGraphicsScreen;
  begin
    Entering('SaveGraphicsScreen');
    {
    Use the heap for the Graphics Screen Buffer Array.
    }
    new(GraphicsScreenPtr);
    {
    Reset the graphics window to the entire screen and save it in an array.
    }
    GraphWindow(0,0,XMaxGlb,YMaxGlb);
    with GraphicsScreenPtr^ do
      GetPic(GraphicsScreenBuffer,0,YMaxGlb,XmaxGlb,0);
    Exiting('SaveGraphicsScreen')
  end;

{-------------------------------------------------------------------------}
```

{.CP18}

```
procedure RestoreGraphicsScreen;
  begin
    Entering('RestoreGraphicsScreen');
    {
    Restore the image to the graphics screen.
    }
    with GraphicsScreenPtr^ do
      PutPic(GraphicsScreenBuffer,0,YMaxGlb);
    {
    Free up the heap area.
    }
    dispose(GraphicsScreenPtr);
    Exiting('RestoreGraphicsScreen')
  end;

{------------------------------------------------------------------------------}
```

{.CP15}

```
procedure SaveOldScreen;
{
PURPOSE: Saves the graphics screen and initializes the text screen.
}
  begin
    Entering('SaveOldScreen');
    SaveGraphicsScreen;
    TextMode(C80);
    TextColor(Cyan);
    TextBackground(Black);
    Exiting('SaveOldScreen');
  end;

{------------------------------------------------------------------------------}
```

{.CP21}

```
procedure RestoreOldScreen;
{
PURPOSE: Restores the graphics screen and resets all low res graphics
parameters.
}
  begin
    Entering('RestoreOldScreen');
    GraphColorMode;
    GraphBackground(Black);
    Palette(3);
    RestoreGraphicsScreen;
    TextXLcl:= TextXGlb;
    TextYLcl:= TextYGlb;
    SelectWindow( WindowNdxGlb );
    TextXGlb:= TextXLcl;
    TextYGlb:= TextYLcl;
    Exiting('RestoreOldScreen');
  end;

{-------------------------------------------------------------------------}
```

{.CP15}

```
procedure Error(ErrorNum: byte);
{
PURPOSE: This routine handles printing error messages to the screen.  The
         messages are printed in white characters on a red background.  If
         the error is an abortive error (i.e., an error which should make
         the program abort) the boolean variable Aborting is set true.
         If the error is not an abortive error, the routine waits for the
         operator to hit escape to acknowlege the error.  If the error is
         not an abortive error and the screen is in graphics mode, the
         total graphics screen is stored in memory and the error is written
         in regular text.  If the error occurs while parsing, the flag
         GoodCommand is set to false to disable taking an action due to the
         command.
}
  const                                                          {.CP15}
    No_Open = 'Could not open';
    Expecting = 'Expecting';
    Invalid = 'Invalid ';
    command_str = ' command';
    beyond = ' would drive motor beyond';
    or_str = ' or';
    motor = ' motor';
    move_value = ' move value';
    not_supported = ' NOT SUPPORTED';
  type
    ErrorTypes = set of byte;
  var
    AbortingErrors, CommandErrors: ErrorTypes;
    ErrM: AnyString;
```

```
begin                                                          {.CP27}
  Entering('Error');
  {
  Set the global flag Aborting if it should be set for this error number.
  }
  AbortingErrors:= [21,22,24];
  Aborting:= (ErrorNum in AbortingErrors);
  {
  Save the previous status of the graphics screen.
  }
  if ((GraphicsOn) and (not Aborting)) then SaveOldScreen;
  {
  Write out the current command for the operator if this is a parsing error.
  }
  CommandErrors:= [1..16,28..36];
  if ErrorNum in CommandErrors
    then
      begin
        writeln('Command: ',CurrentCommand);
        GoodCommand:= false
      end;
  {
  Prepare to write the error message in White on Red.
  }
  TextColor(White);
  TextBackground(Red);
  ClrEol;
  case ErrorNum of                                             {.CP19}
  {
  The first 18 errors are the same as the Hero Compiler errors.
  }
     1: ErrM:= Expecting + ' keyword';
     2: ErrM:= Expecting + ' cont or wait';
     3: ErrM:= invalid + ' speech number';
     4: ErrM:= Expecting + ' semicolon';
     5: ErrM:= invalid + ' graphics parameter';
     6: ErrM:= invalid + ' wrist' + command_str + '; ' +
                Expecting + ' PIVOT' + or_str + ' ROTATE';
     7: ErrM:= invalid + ' arm ' + command_str + '; ' +
                Expecting + ' EXTEND' + or_str + ' SHOULDER';
     8: ErrM:= Expecting + ' CONT' + or_str + ' WAIT';
     9: ErrM:= Expecting + ' ABS' + or_str + ' REL';
    10: ErrM:= Expecting + motor + ' speed';
    11: ErrM:= invalid + motor +  command_str;
    12: ErrM:= invalid + ' absolute ' + motor + move_value;
    13: ErrM:= invalid + ' relative ' + motor + move_value;
```

```
                                                                    {.CP7}
{     14: ErrM:= command_str +  beyond + ' lower limit';
      15: ErrM:= command_str +  beyond + ' upper limit';
      16: ErrM:= Expecting + ' end statement';
      17: ErrM:= invalid + ' memory location...using $0100';
      18: ErrM:= invalid + ' object ' + command_str + '; ' +
                 Expecting + ' CC SS XX';}
                                                                    {.CP25}
{
The errors from 21 on are RIGS errors only.
}
      21: ErrM:= No_Open + ' one of the robot data files.';
      22: ErrM:= No_Open + ' the font file (4x6.FON).';
      23: ErrM:= No_Open + ' source file ' + FileName + '.';
      24: ErrM:= 'Text string sent to DrawText contains illegal characters.';
      25: ErrM:= 'Please respond with Y or N or hit Return for the default.';
      26: ErrM:= 'Robot moved while Steering Wheel is turned;' +
                 not_supported + '!';
      27: ErrM:= No_Open + ' tokens file ' + FileName + '; tokens not saved.';
      28: ErrM:= invalid + ' limits specifications.';
      29: ErrM:= invalid + ' starting point specifications.';
      30: ErrM:= invalid + ' number of parameters.';
      31: ErrM:= 'CONT is' + not_supported + ';  ' +
                 command_str + 'treated as WAIT.';
      32: ErrM:= No_Open + ' log file ' + FileName + '; ' +
                 command_str + 's not logged.';
      33: ErrM:= No_Open + ' background file ' + FileName + '.';
      34: ErrM:= invalid + ' BASE ' + command_str + ' in background file.';
      35: ErrM:= invalid + ' LINE ' + command_str + ' in background file.';
      36: ErrM:= 'Maximum number of background LINES exceeded.';
     else ErrM:= 'This error number is not defined.';
   end;
   writeln('Error ',ErrorNum,': ', ErrM);                           {.CP4}
   if LogCommands
     then writeln(LogFile,'Error ',ErrorNum,': ',ErrM);
   ErrorCount:=ErrorCount+1;
```

```
    if (not Aborting)                                          {.CP19}
      then
        begin { Make the operator acknowledge the error }
          if LogCommands then writeln(LogFile);
          write('Press <Esc> to continue.');
          Beep;
          WaitForEsc;
          if GraphicsOn
            then
              begin
                RestoreOldScreen;
                Beep;
              end
            else
              begin
                TextColor(Cyan);
                TextBackground(Black)
              end;
        end
      else                                                     {.CP13}
        begin
          ErrM:= 'ABORTING!';
          writeln(Errm);
          if LogCommands
            then writeln(LogFile,Errm);
          TextColor(Cyan);
          TextBackground(Black)
        end;
    Exiting('Error');
  end;

{-------------------------------------------------------------------}
                                                               {.CP14}
procedure DrawAscii(var X, Y, AscColor: integer; ch: byte);
{
PURPOSE: Draws an ascii character in the 4x6 font style at the X Y position
and in the color passed.  The X and Y position is the graphics coordinate of
the lower lefthand corner of the characters position.  The color is the
palette color.  The character must be between 32 and 126.  The X and Y
coordinates are updated to those for the next character, and line feeds are
handled if the character is at the right hand edge of the screen.
}
  var
    xpos,ypos,xx,yy: integer;
    charbyte: byte;
```

```
  begin { DrawAscii }                                                {.CP24}
    for ypos:=0 to 5 do
      begin
        CharByte:= (CharFont[ch,(7-ypos) shr 1] shr ((ypos and 1) shl 2))
                   and $0F;
        for xpos:=0 to 3 do
          begin
            xx:=x+xpos;
            yy:=y-4+ypos;
            if ((CharByte shr (3-xpos)) and 1)<>0
              then Plot(xx,yy,AscColor)
              else Plot(xx,yy,0);
          end;
      end;
    x:= x+5;
    if x>XMaxGlb
      then
        begin
          x:=0;
          y:= y + 6;
        end;
  end;

{-------------------------------------------------------------------------}
                                                                   {.CP22}
procedure DrawText( MColor: integer; Message: FontString; Cr: boolean);
{
PURPOSE: Draws a string of characters in the 4x6 font in the color passed.
Scrolling of the window is handled by this routine.

INPUTS: MColor - The palette color for the string Message.
        Message - The string to be written.  Length must be <= 64 characters.
        Cr - A flag which indicates whether a carriage return is desired at
             the end of the string.
}
  type
    ScrollBufHeap = ^ScrollBufRec;
    ScrollBufRec = record
                     ScrollBuffer: array[0..2885] of byte;
                   end;

  var
    AscChar: char;
    AscOrd: byte;
    LetterNum, MesLength: integer;
    ScrollBufPtr: ScrollBufHeap;
```

```
begin                                                          {.CP16}
  {Entering('DrawText');}
  MesLength:= length(Message);
  LetterNum:= 0;
  while ((LetterNum<MesLength) and (not Aborting)) do
    begin
      LetterNum:= LetterNum + 1;
      AscChar:= copy(Message, LetterNum, 1);
      if AscChar in [#32..#126]
        then
          begin
            AscOrd:= ord(AscChar);
            DrawAscii( TextXGlb, TextYGlb, MColor, AscOrd);
          end
        else Error(24);
    end;
  if ((not Aborting) and Cr) then                              {.CP23}
    begin
      TextXGlb:= 0;
      if TextYGlb+6>Y2RefGlb
        then
          begin
            {
            We will scroll the screen.
            }
            New(ScrollBufPtr);
            with ScrollBufPtr^ do
              begin
                GetPic(ScrollBuffer,X1RefGlb,Y2RefGlb,X2RefGlb,Y1RefGlb+6);
                ClearScreen;
                PutPic(ScrollBuffer,X1RefGlb,TextYGlb-6)
              end;
            dispose(ScrollBufPtr);
          end
        else TextYGlb:= TextYGlb + 6;
    end;
  {Exiting('DrawText')}
end;

{-------------------------------------------------------------------------}
```

{.CP17}

```
function YesNo(Question:AnyString; Default:boolean):boolean ;
{
  PURPOSE: handles displaying questions and receiving a response of either
           yes or no from the operator.  This routine keeps updating the
           time while waiting for the operator.
  INPUT:
    Question - a string of any length which is the question to be asked.
    Default - the default answer, yes or no, passed using the global variables
              yes or no.
  OUTPUT: returns true if yes or false if no.
  EXTERNAL PROCEDURES CALLED:
    Beep
    Error
}
  var                                                              {.CP8}
    Response: char;
    Defstr: char;
    DidIt: boolean ;
    Qmess: FontString;
    Qc: integer;

  begin; { YesNo }                                                 {.CP22}
    if Default
      then Defstr:='Y'
      else Defstr:='N';
    if GraphicsOn
      then
        begin
          Qc:= 1;
          Qmess:= Question + ' (' + Defstr + ') ? ';
          Qmess:= Upc(Qmess);
          DrawText(Qc, Qmess, No)
        end;
    repeat
      if not GraphicsOn then
        begin
          writeln;
          write(Question,' (Y/N, default=',Defstr,') ? ')
        end;
      Beep ;
      repeat until KeyPressed;
      read(Kbd,Response) ;
      Response := UpCase(Response) ;
```

```
    case Response of                                                    {.CP17}
      #13: begin { Return was hit...take the default answer. }
             DidIt := true ;
             Yesno := Default
           end;
      'Y': begin
             DidIt:=true ;
             if GraphicsOn
               then
                 begin
                   Qmess:=Upc('Yes');
                   DrawText(Qc, Qmess, Yes)
                 end
               else write('Yes');
             Yesno:=true
           end ;
      'N': begin                                                        {.CP20}
             DidIt:=true ;
             if GraphicsOn
               then
                 begin
                   Qmess:=Upc('No');
                   DrawText(Qc, Qmess, Yes)
                 end
               else write('No');
             Yesno:=false
           end ;
        else
          begin
            DidIt:=false ;
            Error(25)
          end ;
    end ;
  until DidIt ;
end ;

{-------------------------------------------------------------------------}
```

```
                                                                    {.CP21}
procedure MatMult4x4( A, B: TransformMatrix );
{
PURPOSE: Matrix Multiplies two four by four matrices and stores the result in
the global matrix Tfm.
}
  var
    I,J,K : integer ;
  begin
    Entering('MatMult4x4');
    for I:=1 to WidthArray do
      for J:=1 to WidthArray do
        begin
          Tfm[I,J]:= 0 ;
          for K:=1 to WidthArray do
            Tfm[I,J] := Tfm[I,J] + A[I,K] * B[K,J]
        end;
    Exiting('MatMult4x4');
  end ;

{-------------------------------------------------------------------------}
                                                                    {.CP18}
procedure NewExt(Ext: string3);
{
PURPOSE: Changes the file extension of the file name FileName to that of the
extension passed.
}
  var
    DotPos, FNlength: integer;
  begin
    Entering('NewExt');
    DotPos:= pos('.', FileName);
    FNlength:= length(FileName);
    delete(FileName, DotPos + 1, FNlength - DotPos);
    FileName:= FileName + Ext;
    Exiting('NewExt')
  end;

{-------------------------------------------------------------------------}
```

```
{-------------------------------------------------------------------------
********** PROGRAM BEGINNING **********
-------------------------------------------------------------------------}

procedure Initialize;
{
PURPOSE: Initializes program variables, draws a welcome screen, reads robot
part data files, gets the source file name and opens it, and, if one is
desired, opens the tokens file.
}

  procedure InitVariables;
  {
  Initializes the program variables and the robot position and limits
  variables.
  }
    var
      J: RobotParts;
    begin
      Debug:= false;
      Debug_Flow:= false;
      if ParamCount>1
        then if ParamStr(2) = 'T' then Debug:=true;
      Echo:=  {true} false ;
      Entering('Initialize');
      Entering('InitVariables');
      {
      Initialize some of the program variables
      }
      Aborting:= false;
      Done := false ;
      ScreenDistance:= 400.0;
      CenterCol:= ScreenCols shr 1;
      ErrorCount:= 0;
      SourceOpen:= false;
      LogCommands:= false;
      TokenOpen:= false;
      SaveTokens:= true;
      AddBackgroundFlag:= false;
      NumBgLines:= 0;
```

{.CP42}

```
{
Provide initial values for the robot parts.  These values correlate
to the settings of the Hero after the initialize command.
}
for J:=Steering to Gripm do
  with Parts[J] do
    begin

      { Initial Position }
      case J of
        Head, Wrrot: Position:= 175.0;
        Wrpvt      : Position:= 90.0;
        else         Position:= 0.0;
        end;

      { Absolute Minimums }
      case J of
        Steering, WrPvt: Absolutes[1]:= -90.;
        Body           : Absolutes[1]:= -360.;
        else             Absolutes[1]:= 0.;
        end;

      {
      Set all of the recalculate flags so the transform matrices will be
      calculated the first time.
      }
      RecalculateFlag:= true;
    end;

  { Absolute Maximums }
  Parts[Steering].Absolutes[2]:= 90.;
  Parts[Body].Absolutes[2]:= 360.;
  Parts[Head].Absolutes[2]:= 350.;
  Parts[Arm].Absolutes[2]:= 150.;
  Parts[ExtendArm].Absolutes[2]:= 5.;
  Parts[WrPvt].Absolutes[2]:= 90.;
  Parts[WrRot].Absolutes[2]:= 350.;
  Parts[Gripp].Absolutes[2]:= 3.5;
  Parts[Gripm].Absolutes[2]:= 3.5;

  Exiting('InitVariables')
end;
```

```
procedure Welcome;                                                    {.CP45}
{
PURPOSE: Writes a Welcome screen on the display.
}
  const
    WelcomeLength = 30;
    Side: char = #219;
  type
    WelcomeStr = string[WelcomeLength];
  var
    WelcomeTop, WelcomeBottom, WelcomeBlank: WelcomeStr;
    I: integer;
  begin
    Entering('Welcome');
    CrtInit ;
    ClrScr ;
    TextMode(C80);
    GraphicsOn:= false;
    TextColor(Cyan);
    WelcomeTop:= '';
    for I:=1 to WelcomeLength do
      WelcomeTop:= Concat(WelcomeTop, chr(220));
    WelcomeBottom:= '';
    for I:=1 to WelcomeLength do
      WelcomeBottom:= Concat(WelcomeBottom, chr(223));
    WelcomeBlank:= Side + '                              ' + Side;
    Window(25,6,55,19);
    GotoXY(1,1);
    writeln(WelcomeTop);
    writeln(WelcomeBlank);
    writeln(Side,'     H E R O  R O B O T     ',Side);
    writeln(WelcomeBlank);
    writeln(Side,'     Graphics Simulator     ',Side);
    writeln(WelcomeBlank);
    writeln(Side,'         Version 3.0        ',Side);
    writeln(WelcomeBlank);
    writeln(WelcomeBlank);
    writeln(Side,'       by Ken Sizemore      ',Side);
    writeln(WelcomeBlank);
    writeln(WelcomeBottom);
    Window(1,1,80,25);
    writeln;
    Exiting('Welcome')
  end;
```

```
procedure ReadDataFiles;                                        {.PA}
{
PURPOSE: Reads the point, line and plane data from each of the Robot's
part data files.
}
  const
    DataFileName: array[RobotParts] of string[80] =
      ('\DATA\MOVWHEEL.DTA', '\DATA\BODY.DTA',   '\DATA\HEAD.DTA',
       '\DATA\ARM.DTA',      '\DATA\EXTARM.DTA', '\DATA\WRPVT.DTA',
       '\DATA\WRROT.DTA',    '\DATA\GRIPP.DTA',  '\DATA\GRIPM.DTA');

  var
    DataFile : text ;
    TempPoint : real ;
    XYZ0 : Coordinate ;
    L : EndPoint;
    TempPlane: Plane;
    I,K, IntPFill, RefNumPoints, RefNumLines, RefNumPlanes : integer ;
    J: RobotParts;
    AllDataRead: boolean;

  begin
    Entering('ReadDataFiles');
    writeln;
    writeln('Reading Robot Part Data Files...please be patient.');
    AllDataRead:= false;
    NumPoints:= 0;
    NumLines:= 0;
    NumPlanes:= 0;
    J:=steering;
    repeat
      if not OpenText(DataFile, DataFileName[J], true)
        then Error(21)
        else
          begin
            if Debug_flow
              then writeln(lst,'file',DataFileName[J]);
            {
            Read the Number of points, Lines and Planes in the file.
            }
            read(DataFile, RefNumPoints, RefNumLines, RefNumPlanes) ;
            {
            Read in the point data, appending it to other points.
            }
            XYZ0[4] := 1 ;
            for I:=1 to RefNumPoints do
              begin
                read(DataFile,XYZ0[1],XYZ0[2],XYZ0[3]) ;
                Points[NumPoints+I] := XYZ0 ;
              end ;
```

{.CP14}

```
{
Read in the line data, appending it to other line data.
Adjust the point indices by NumPoints.
}
for I:=1 to RefNumLines do
  with Lines[NumLines+I] do
    begin
      read(DataFile,L[1],L[2]) ;
      for K:=1 to 2 do
        if L[K]>0
          then EndPts[K]:= L[K] + NumPoints
          else EndPts[K]:= L[K] + NumPoints + 1;
    end;
```

{.CP31}

```
{
Read in the plane data, appending it to other plane data.
Adjust the point indices by NumPoints and the line indices
by NumLines.
}
for I:=1 to RefNumPlanes do
  begin
    with TempPlane do
      begin
        for K:=1 to WidthArray do
          read(DataFile, Vertices[K]);
        for K:=1 to WidthArray do
          read(DataFile, Edges[K]);
        read(DataFile, IntPFill);
        PFill:= boolean(IntPFill);
        for K:= 1 to WidthArray do
          if Vertices[K]>0
            then Vertices[K]:= Vertices[K] + NumPoints
            else
              if Vertices[K]<>0
                then Vertices[K]:= Vertices[K] + NumPoints + 1;
        for K:= 1 to WidthArray do
          if Edges[K]>0
            then Edges[K]:= Edges[K] + NumLines
            else
              if Edges[K]<> 0
                then Edges[K]:= Edges[K] + NumLines + 1;
      end;
    Planes[NumPlanes+I]:= TempPlane;
  end;
```

{.CP25}

```
            {
            Update the Robot Record and the Globals.
            }
            with Parts[J] do
              begin
                Index[1].First:= NumPoints+1;
                Index[1].Last:= NumPoints + RefNumPoints;
                Index[2].First:= NumLines+1;
                Index[2].Last:= NumLines + RefNumLines;
                Index[3].First:= NumPlanes+1;
                Index[3].Last:= NumPlanes + RefNumPlanes;
              end;
            NumPoints:= NumPoints + RefNumPoints;
            NumLines:= NumLines + RefNumLines;
            NumPlanes:= NumPlanes + RefNumPlanes;
            close(DataFile);
          end;
        if (J = gripm)
          then AllDataRead:= true
          else J:=Succ(J);
      until ((Aborting) or (AllDataRead));

      Exiting('ReadDataFiles')
    end;

procedure ReadFontFile;                                          {.CP18}
{
PURPOSE: Reads the font file 4x6.FON into the array for the font data.
}
  var
    FontFil: file of CharArray;
    FontOK: boolean;
  begin
    Entering('ReadFontFile');
    assign(FontFil,'4X6.FON');
    {$I-} reset(FontFil); {$I+}
    FontOK:= (IOresult = 0);
    if FontOK
      then read(FontFil,CharFont)
      else Error(22);
    close(FontFil);
    Exiting('ReadFontFile')
  end;
```

```
procedure GetSourceName;                                        {.PA}
{
PURPOSE: Determines the Source file name and opens it.  If the runstring
contains a parameter, the file name is set to that string and the program
tries to open that file name.  If no runstring parameter was passed or the
file did not exist, the operator is prompted to enter the source file name.
}
  var
    FileOK, ParamFlag, SemiAborting: boolean;
  begin
    Entering('GetSourceName');
    ParamFlag:= (ParamCount>0);
    repeat
      if ParamFlag
        then
          begin
            ParamFlag:= false;
            SemiAborting:= false;
            FileName:= ParamStr(1);
            FileName:= Upc(FileName);
            if (pos('.', FileName) = 0)
              then FileName:= FileName + '.SRC';
          end
        else
          begin
            repeat
              Beep;
              writeln;
              write('Enter the name of the Source file: ');
              readln(FileName);
              writeln;
              SemiAborting:= (Length(FileName)=0);
              if SemiAborting
                then writeln('You want to Abort the program.')
                else
                  begin
                    FileName:= Upc(FileName);
                    if (pos('.', FileName) = 0)
                      then FileName:= FileName + '.SRC';
                    writeln('Source file name is ',FileName,'.');
                  end;
            until YesNo('Is this correct',Yes);
          end;
```

```
    if SemiAborting                                                {.CP18}
      then Aborting:= true
      else
        begin
          FileOK:= OpenText(Source,FileName, true);
          if not FileOK
            then Error(23)
            else SourceOpen:= true;
        end;
  until FileOK or Aborting;
  if not Aborting then
    begin
      readln(Source,NextCommand);
      NextCommand:= Upc(NextCommand);
    end;
  Exiting('GetSourceName')
end;

procedure StartLogFile;                                            {.CP19}
{
PURPOSE: Opens a file to log the commands and errors from this program.  If
the log file cannot be opened, an error message is written and the
program proceeds.
}
  begin
    Entering('StartLogFile');
    NewExt('LOG');
    if not OpenText(LogFile, FileName, false)
      then
        begin
          LogCommands:= false;
          close(LogFile);
          Error(32)
        end
      else LogCommands:= true;
    Exiting('StartLogFile')
  end;
```

```
procedure StartTokenFile;                                        {.CP19}
{
PURPOSE: Opens a file to store the tokens parsed by this program.  If
the tokens file cannot be opened, an error message is written and the
program proceeds.
}
  begin
    Entering('StartTokenFile');
    NewExt('GTK'); { for Graphics Tokens }
    if not OpenText(TokFile, FileName, false)
      then
        begin
          SaveTokens:= false;
          close(TokFile);
          Error(27)
        end
      else TokenOpen:= true;
    Exiting('StartTokenFile')
  end;

begin                                                            {.CP10}
  InitVariables;
  Welcome;
  ReadDataFiles;
  if not Aborting then ReadFontFile;
  if not Aborting then GetSourceName;
  if not Aborting then StartLogFile;
  if SaveTokens and not Aborting then StartTokenFile;
  Exiting('Initialize');
end;

{-------------------------------------------------------------------}
```

```
procedure ReadAndDraw;
{
PURPOSE: This procedure will accept commands from either the Source file or
the operator and draw the updated views of the robot on the screen.  The
views of the robot shown are an Overhead view and a Shoulder view.
}
  const
    GlbStatusX = 0;
    GlbStatusY = 5;
    identity: TransformMatrix = (( 1.0, 0.0, 0.0, 0.0 ),
                                 ( 0.0, 1.0, 0.0, 0.0 ),
                                 ( 0.0, 0.0, 1.0, 0.0 ),
                                 ( 0.0, 0.0, 0.0, 1.0 ));
  var
    CphiArm, SphiArm: real;
    ArmLeft, DrawShoulderViewFlag: boolean;

{-------------------------------------------------------------------------}
                                                                {.CP21}
procedure AddPerspective(var Ppt: Coordinate) ;
{
PURPOSE: This procedure changes the X and Y values (the values to be plotted
later) to give the drawing perspective.  This is done by multiplying each
X and Y value by the distance from the viewers eye divided by the value of
the Z coordinate for that point.
}
  var
    Perspective, Z: real;
    I, J: integer;
  begin
    Z:= abs(Ppt[3]);
    if (Z=0)
      then Perspective:= ScreenDistance/(0.001)
      else Perspective:= ScreenDistance/Z;
    for J:=1 to 2 do
      Ppt[J]:= Ppt[J] * Perspective;
  end;

{-------------------------------------------------------------------------}
```

```
                                                                    {.CP16}
procedure FindHiddenLines;
{
PURPOSE: This procedure determines which lines of the drawing are hidden
from view.  Since the viewer is looking down the Z axis at the shape, he
is looking in a positive Z direction.  Considering the plane equation for
each of the planes in the shape,  if the plane is facing the 0,0,0 point
(toward the viewer), then for a point on that plane the value of C is
negative.  Likewise, any point where the value of C is positive faces away
from the viewer and would not be seen.  This works for only convex surfaces.
}
  var
    XYZ: array[ArrayWidth] of Coordinate;
    ZeroCoord: Coordinate;
    A,B,C,Unit:real;
    I, J: integer;
  begin                                                             {.CP6}
    Entering('FindHiddenLines');
    DrawText(1,'DETERMINING HIDDEN LINES.',Yes);
    NumSortPlanes:= 0; {Ch}
    for I:=1 to WidthArray do
      ZeroCoord[I]:= 0;
    for I:=1 to NumPlanes do                                        {.CP23}
      begin
        {
        Determine for each plane the Z vector component of its
        plane equation.  This value is C.
        }
        with Planes[I] do
          for J:=1 to WidthArray do
            if Vertices[J]<>0
              then XYZ[J]:= ViewPts[View].Vpts[Vertices[J]]
              else XYZ[J]:= ZeroCoord;
        A:= XYZ[1,2] * (XYZ[2,3] - XYZ[3,3]) +
            XYZ[2,2] * (XYZ[3,3] - XYZ[1,3]) +
            XYZ[3,2] * (XYZ[1,3] - XYZ[2,3]);
        B:= -(XYZ[1,1] * (XYZ[2,3] - XYZ[3,3]) +
              XYZ[2,1] * (XYZ[3,3] - XYZ[1,3]) +
              XYZ[3,1] * (XYZ[1,3] - XYZ[2,3]));
        C:= XYZ[1,1] * (XYZ[2,2] - XYZ[3,2]) +
            XYZ[2,1] * (XYZ[3,2] - XYZ[1,2]) +
            XYZ[3,1] * (XYZ[1,2] - XYZ[2,2]);
        Unit:= Sqrt(Sqr(A)+Sqr(B)+Sqr(C));
        if (Unit=0) then Unit:= 0.001;
        C:= C / Unit;
```

```
                                                                    {.CP27}
      { Only the planes whose C value is negative are facing us.  If the
      plane is visible, store its index in the sorted plane array. }
      if (C < 0)
        then
          begin
            NumSortPlanes:= NumSortPlanes + 1;
            with SortPlane[NumSortPlanes] do
              begin
                PlaneIndex:= I;
                FillFlag:= (C < -0.15) and (Planes[I].PFill);
                {
                Determine the minimum Z value for sorting purposes.
                }
                MinZ:= XYZ[1,3];
                for J:=2 to WidthArray do
                  if (( XYZ[J,3] < MinZ ) and ( XYZ[J,3] > 0))
                    then MinZ:= XYZ[J,3];
              end;
          end;
      if Echo then
        writeln(lst,'Plane[',I,']=',C:3:4);
    end;
  Exiting('FindHiddenLines');
end;

{-------------------------------------------------------------------------}
```

{.CP34}

```
procedure SortPlanes;
{
PURPOSE: Sort the variables in the array SortPlane descending from the
maximum value of the MinZ variable.
}
  var
    I, J, K: integer;
    NoSwap: boolean;
    TempSort: SortRec;
  begin
    Entering('SortPlanes');
    DrawText(1,'SORTING PLANE DATA.',Yes);
    repeat
      NoSwap:= true;
      for I:=1 to NumSortPlanes-1 do
        if SortPlane[I].MinZ < SortPlane[I+1].MinZ
          then
            begin
              NoSwap:= false;
              TempSort:= SortPlane[I];
              SortPlane[I]:= SortPlane[I+1];
              SortPlane[I+1]:= TempSort
            end;
    until NoSwap;
    {
    for I:=1 to NumSortPlanes do
      with SortPlane[I] do
        writeln(I,': MinZ=',MinZ,' PlaneIndex=',PlaneIndex);
    }
    Exiting('SortPlanes');
  end;

{-------------------------------------------------------------------------}
```

{.CP37}

```
procedure MaxMin ;
{
PURPOSE: Determines the maxima and minima of the X and Y coordinates in the
array Vpts for the current View.
}
  var
    X,Y : real ;
    I : integer ;
  begin ;
    Entering('MaxMin');
    DrawText(1,'DETERMINING MAXIMA AND MINIMA.',Yes) ;
    with ViewPts[View] do
      begin
        Xmin := Vpts[1,1] ;
        Xmax := Vpts[1,1] ;
        Ymin := Vpts[1,2] ;
        Ymax := Vpts[1,2] ;
        for I:= 2 to NumPoints do
          begin
            X := Vpts[I,1] ;
            Y := Vpts[I,2] ;
            if X<Xmin then Xmin:=X ;
            if X>Xmax then Xmax:=X ;
            if Y<Ymin then Ymin:=Y ;
            if Y>Ymax then Ymax:=Y ;
          end ;
        if Echo then
          begin
            writeln(lst,'Xmin = ',Xmin,' Ymin = ',Ymin) ;
            writeln(lst,'Xmax = ',Xmax,' Ymax = ',Ymax) ;
          end ;
      end;
    Exiting('MaxMin');
  end ;

{-------------------------------------------------------------------------------}
```

{.CP31}

```
procedure Scale ;
{
PURPOSE: Sets up the scaling factors based upon the minimum and maximum
values of the X and Y data.  The scaling factors will be used to convert from
world (user) coordinates to window (screen) coordinates.  The current window
coordinates, stored in the global X and Y reference variables, will be used.

INPUTS: Xmin, Ymin, Xmax, Ymax, X2RefGlb, Y2RefGlb
OUTPUTS: Xmin, Ymin, Xmax, Ymax (modified)
         Xslope, Xint, Yslope, Yint
}
  var
    XYAspectRatio, Xrange, Yrange, Range, Halfrange, Xmid, Ymid : real ;
  begin ;
    Entering('Scale') ;
    {
    Determine the Aspect Ratio of the screen.  The window must have been
    selected in order for this to work.  Note that the integer Global
    window coordinates are converted to real numbers using the 'int'
    function (which always returns a real).
    }
    XYAspectRatio:= (5.0 * (int(X2RefGlb)+1.0)) / (6.0 * (int(Y2RefGlb)+1.0));
    if Echo then writeln(lst,'XYAspectRatio = ',XYAspectRatio);
    {
    Determine the ranges of X and Y, then their midpoints.
    }
    Xrange := abs(Xmax-Xmin) ;
    Yrange := abs(Ymax-Ymin) ;
    Xmid := Xmin + 0.5*Xrange ;
    Ymid := Ymin + 0.5*Yrange ;
```

{.CP19}

```
    {
    Adjust the Y range by the AspectRatio, then compare the values.  Add
    ten percent (for margins) to the total range.
    }
    Yrange := Yrange * XYAspectRatio;
    if Xrange>Yrange
      then Range:=1.1*Xrange
      else Range:=1.1*Yrange ;
    Halfrange := 0.5*Range ;
    Xmin := Xmid - Halfrange ;
    Xmax := Xmid + Halfrange ;
    Ymin := Ymid - ( Halfrange / XYAspectRatio );
    Ymax := Ymid + ( Halfrange / XYAspectRatio );
    if Echo then
      begin
        writeln(lst,'Xmin = ',Xmin,' Ymin = ',Ymin) ;
        writeln(lst,'Xmax = ',Xmax,' Ymax = ',Ymax) ;
      end ;
```

```
                                                                    {.CP19}
    {
    The straight line equation formulas are used to find the scale factors.
    Note that the integer Global window coordinates are converted to real
    numbers using the 'int' function (which always returns a real).
    }
    Xslope := int(X2RefGlb) / Range ;
    Xint := -Xslope * Xmin ;
    Yslope := int(Y2RefGlb) * XYAspectRatio / Range ;
    Yint := -Yslope * Ymin ;
    if Echo then
      begin
        writeln(lst,'Xslope = ',Xslope,' Xint = ',Xint) ;
        writeln(lst,'Yslope = ',Yslope,' Yint = ',Yint) ;
      end ;
    Exiting('Scale') ;
  end ;

{-------------------------------------------------------------------------}
                                                                    {.CP14}
procedure PlotPlanes ;
{
PURPOSE: Plots the planes whose indices are stored in the array SortPlane.
In order to hide previously drawn lines behind the plane, the outline of
the planes are drawn first in light blue, filled with purple, filled with the
background color, then redrawn with white.
}
  type
    ScXY = array[1..2] of integer ;
  var
    ScPts:array[MaxPoints] of ScXY ;
    CenterX, CenterY, I, J, K, P1, P2 : integer ;

  procedure PlotEdges(SortedPlaneNumber, EdgeColor: integer);        {.CP18}
  {
  PURPOSE:  Draws the lines which outline a plane in the color passed.
  }
    begin
      with Planes[SortPlane[SortedPlaneNumber].PlaneIndex] do
        for J:= 1 to WidthArray do
          begin
            K := Edges[J];
            if K<>0 then
              begin
                P1 := Lines[K].EndPts[1] ;
                P2 := Lines[K].EndPts[2] ;
                Draw(ScPts[P1,1], ScPts[P1,2],
                     ScPts[P2,1], ScPts[P2,2], EdgeColor);
              end;
          end ;
    end;
```

```
procedure FillPlane(SortedPlaneNumber: integer);                    {.CP25}
{
PURPOSE:  In order to fill the planes, a center point of the planes is
determined by summing all the half vectors from one point to the next.
This works for convex plane shapes only.
}
  var
    TempVert:integer;
  begin
    with Planes[SortPlane[SortedPlaneNumber].PlaneIndex] do
      begin
        CenterX:= ScPts[Vertices[1],1];
        CenterY:= ScPts[Vertices[1],2];
        for J:=2 to WidthArray do
          begin
            TempVert:= Vertices[J];
            if TempVert <> 0 then
              begin
                CenterX:= CenterX + ((ScPts[TempVert,1] - CenterX) div 2);
                CenterY:= CenterY + ((ScPts[TempVert,2] - CenterY) div 2)
              end
          end;
      end;
  end;

begin ;                                                             {.CP20}
  Entering('PlotPlanes') ;
  {
  Change the points into screen units using the scaling factors.
  }
  with ViewPts[View] do
    for I:=1 to NumPoints do
      begin
        ScPts[I,1] := round(Xslope * Vpts[I,1] + Xint) ;
        ScPts[I,2] := round(Yslope * Vpts[I,2] + Yint) ;
      end ;
  if Echo then
    for I:=1 to NumPoints do
      writeln('ScPts[',I,'] = ',ScPts[I,1],',',ScPts[I,2]) ;
  if (View=Shoulder) or (not AddBackgroundFlag) then ClearScreen;
  {
  Plot only the edges of the first plane.
  }
  I:=1;
  PlotEdges(I,3);
```

{.CP24}

```
{
Plot the planes by drawing the edges in magenta, filling the plane
with magenta to cover the cyan background lines and the white plane
lines, redraw the edges in white, and fill the plane with black.
If the fill flag for the plane is false, just plot the edges in white.
}
for I:=2 to NumSortPlanes do
  begin
    if SortPlane[I].FillFlag
      then
        begin
          PlotEdges(I,2);
          FillPlane(I);
          FillShape(CenterX,CenterY, 2, 2);
          PlotEdges(I,3);
          FillShape(CenterX,CenterY, 0, 3)
        end
      else PlotEdges(I,3);
  end;
 Exiting('PlotPlanes') ;
end ;

{-------------------------------------------------------------------------}
```

```
                                                                {.CP31}
procedure RandDinit;
{
PURPOSE: Initialize the variables used in the program which are declared in
procedure ReadAndDraw (i.e., non-Global variables).
}
  var
    I: integer;
    J: RobotParts;
    PhiArm: real;
  begin
    Entering('RandDinit');
    DrawShoulderViewFlag:= true;
    ArmLeft:= false;
    {
    Initialize the variables used in windowing.
    }
    InitGraphic;
    {
    Calculate the angle of rotation around the y-axis of the robot's arm.
    }
    PhiArm:= arctan( 1.0 / 11.0 );
    CphiArm:= cos(PhiArm);
    SphiArm:= sin(PhiArm);
    {
    Initialize the robot's current position.
    }
    for I:=1 to 2 do
      RobotPos[I]:= 0;
    RobotPos[3]:= ScreenDistance;
    NewRobotPos:= true;
    Exiting('RandDinit')
  end;
```

{.CP34}

```
procedure ScreenInit;
{
PURPOSE: Sets up the screen for low resolution graphics and defines the four
windows used in ReadAndDraw.
}
  begin
    Entering('ScreenInit');
    {
    Initialize the screen for the low resolution graphics.  Low res must
    be used so the hidden line algorithms will work.
    }
    GraphColorMode;
    GraphWindow(0,0,XMaxGlb,YMaxGlb);
    Palette(3);
    GraphBackground(0);
    TextColor(1);
    GraphicsOn:= true;
    {
    Create the windows.
    }
    DefineWindow(1,1,8,200,172);       { Overhead View }
    DefineWorld(1, -30.0, -30.0, 30.0, 30.0);
    DefineWindow(2,202,8,318,172);     { Shoulder View }
    DefineWindow(3,0,176,319,199);     { Comment Window }
    DefineWindow(4,0,0,319,5);         { Status Line Window }
    {
    Draw Borders around the view windows.
    }
    DrawBorder(1);
    DrawBorder(2);
    SelectWindow(3);
    ClearScreen;
    Exiting('ScreenInit')
  end;
```

```
                                                                    {.CP16}
procedure UpdateStatusLine;
{
PURPOSE: This procedure updates the status line located at the top of the
screen in window 4.  The status line contains the current X and Y positions
of the robot and the positions of all the moving parts of the robot.
}
  const
    Heading: array[RobotParts] of string[2] =
      ('ST', 'BD', 'HD', 'SH', 'AR', 'WP', 'WR', 'GR', 'GR');
  type
    string4 = string[4];
  var
    StC: integer;
    StatusMess: FontString;
    J: RobotParts;
    ChangeValStr: string4;

  procedure EliminateBlanks(var ValStr: string4);                   {.CP16}
  {
  PURPOSE: Changes blanks to zeros.  If there is a minus sign in the string,
  the blanks are left alone.
  }
    var
      BlankPosition: integer;
    begin
      BlankPosition:= pos(' ', ValStr);
      while ((BlankPosition<>0) and (pos('-', ValStr)=0)) do
        begin
          delete(ValStr, BlankPosition, 1);
          insert('0',ValStr, BlankPosition);
          BlankPosition:= pos(' ', ValStr)
        end;
    end;
```

```
begin                                                          {.CP32}
  Entering('UpdateStatusLine');
  TextXLcl:= TextXGlb;
  TextYLcl:= TextYGlb;
  SelectWindow(4);
  ClearScreen;
  StC:= 1;
  str((RobotPos[1]/12.0):4:1, ChangeValStr);
  EliminateBlanks(ChangeValStr);
  StatusMess:= 'X' + ChangeValStr;
  str((RobotPos[2]/12.0):4:1, ChangeValStr);
  EliminateBlanks(ChangeValStr);
  StatusMess:= StatusMess + ' Y' + ChangeValStr;
  for J:=Steering to Gripp do
    begin
      case J of
        ExtendArm, Gripp:
          str(Parts[J].Position:3:1, ChangeValStr);
        Body:
          str(Parts[J].Position:4:0, ChangeValStr);
        else
          str(Parts[J].Position:3:0, ChangeValStr);
        end;
      EliminateBlanks(ChangeValStr);
      StatusMess:= StatusMess + ' ' + Heading[J] + ChangeValStr
    end;
  DrawText(StC, StatusMess, No);
  SelectWindow(3);
  TextXGlb:= TextXLcl;
  TextYGlb:= TextYLcl;
  Exiting('UpdateStatusLine')
end;
```

```
procedure ReadSourceCode;                                        {.PA}
{
PURPOSE: Reads and parses the Hero Robot Language commands and sets up the
variables for plotting.  The source code may from the source file or may be
entered by the operator.
}
  var
    RbtPrt: RobotParts;
    GoodResponse: boolean;

  procedure GetSource;
  {
  PURPOSE: Gets a line of source code from either the Source file or from
  the operator.  The command is stored in the variable CurrentCommand.
  The next line of the source file is read into the variable NextCommand.
  }
    var
      TypingCommand: boolean;
      I: integer;

    procedure PromptSource;                                      {.CP18}
    {
    PURPOSE: Prints the next line of the source file and prompts the operator
    to enter a return to execute that command, hit the escape key to exit,
    or hit any other key to type in a command.
    }
      begin
        Entering('PromptSource');
        DrawText(1,'THE NEXT SOURCE FILE LINE IS',Yes);
        DrawText(2,NextCommand,Yes);
        DrawText(1,'HIT ',No);
        DrawText(3,'RETURN ',No);
        DrawText(1,'TO EXECUTE SOURCE LINE, ',No);
        DrawText(3,'ESCAPE ',No);
        DrawText(1,'TO EXIT,',Yes);
        DrawText(1,'OR ANY OTHER KEY TO TYPE IN A COMMAND.',No);
        Beep;
        Exiting('PromptSource');
      end;
```

```
procedure ReceiveSourceResponse;                                {.CP23}
{
PURPOSE: Receives the response to the question in PromptSource and
sets the flag TypingCommand to true if the operator wants to type
in a command, set Done to true if the operator hits the Escape key,
or sets both flags to false if the return key is hit.
}
  var
    ResponseCh: char;
  begin
    Entering('ReceiveSourceResponse');
    repeat until KeyPressed;
    read(kbd,ResponseCh);
    TypingCommand:= false;
    case ResponseCh of
      #13:  TypingCommand:= false;
      #27:  if KeyPressed
              then TypingCommand:= true {for Function and Arrow Keys}
              else Done:= true;
      else  TypingCommand:= true;
    end;
    Exiting('ReceiveSourceResponse');
  end;
```

```
procedure DisplayChoices;                                    {.CP32}
{
Displays all of the Robot Language Commands.
}
  const
    Wc = '  (Wait/Cont)';
    Ar = '  (Abs/Rel)';
    Smf = '  (Slow/Med/Fast)';
    Dg = '  <degrees>';
    Inch = '  <inches>';

  begin
    Entering('DisplayChoices');
    writeln('                          H.I.C.L.A.S.S.  LANGUAGE COMMANDS');
    writeln;
    writeln('Initialize;');
    writeln('Memory  $<hex 4 digit address>;');
    writeln('Speak', Wc, '  <integer phrase number>;');
    writeln('GRAphics  Background    (OFF/<filename>);');
    writeln('   "      Limits        <x1pos> <y1pos> <x2pos> <y2pos>;');
    writeln('   "      ReverseArm;');
    writeln('   "      Startingpoint <xpos>  <ypos>',Dg,';');
    writeln;
    writeln('Arm  Extend  ', Wc, Ar, Smf, Inch);
    writeln(' "   Shoulder', Wc, Ar, Smf,  Dg);
    writeln('Drive        ', Wc, '           ', Smf,'  <feet>');
    writeln('GRIpper      ', Wc, Ar, Smf, Inch);
    writeln('Head         ', Wc, Ar, Smf,  Dg);
    writeln('Turn         ', Wc, Ar, Smf,  Dg);
    writeln('Wrist Pivot  ', Wc, Ar, Smf,  Dg);
    writeln('  "   Rotate ', Wc, Ar, Smf,  Dg);
    writeln;
    Exiting('DisplayChoices')
  end;
```

```
begin                                                          {.PA}
  Entering('GetSource');
  repeat
    PromptSource;
    ReceiveSourceResponse;
    DrawShoulderViewFlag:= true;
    NewRobotPos:= false;
    GoodResponse:= false;
    if Done then
      begin
        if LogCommands then
          writeln(LogFile,'Exit Selected.');
      end
    else
      begin
        for RbtPrt:=Steering to Gripm do
          with Parts[RbtPrt] do
            RecalculateFlag:= false;
        BodyIncrement:= 0;
        if not TypingCommand
          then
            begin
              CurrentCommand:= NextCommand;
              GoodResponse:=true;
              if not eof(Source)
                then
                  begin
                    readln(Source, NextCommand);
                    NextCommand:= Upc(NextCommand)
                  end
                else NextCommand:= 'END;';
            end
```

```
          else                                                    {.CP28}
            begin
              SaveOldScreen;
              GraphicsOn:= false;
              DisplayChoices;
              writeln('Enter a Hero Robot Language Command: ',
                      '( <= 64 characters) ');
              TextBackground(Cyan);
              TextColor(Blue);
              for I:=1 to FontStrLen do
                write(' ');
              GotoXY(1, WhereY);
              readln(CurrentCommand);
              TextColor(Cyan);
              TextBackground(Black);
              CurrentCommand:= Upc(CurrentCommand);
              GoodResponse:= (Length(CurrentCommand) <> 0);
              GraphicsOn:= true;
              RestoreOldScreen;
            end;
        DrawText(1,' ',Yes);
      end;
  until GoodResponse or Done or Aborting;
  CommandPosition:=0;
  CommandLength:= Length(CurrentCommand);
  if (LogCommands and (not Done) and (not Aborting))
    then writeln(LogFile, CurrentCommand);
  Exiting('GetSource');
end;
```

```
procedure Parser;                                                    {.CP6}
{
PURPOSE: Parses the Hero Language Command stored in the string
CurrentCommand and sets variables used in plotting according to the
command received.
}
  procedure PressAnyKey;                                             {.CP20}
  {
  This procedure prompts the operator from the graphics screen to
  press any key to continue.
  }
    var
      Key: char;
    begin
      DrawText(1,'PRESS ',No);
      DrawText(3,'ANY KEY ',No);
      DrawText(1,'TO CONTINUE:',No);
      Beep;
      repeat until keypressed;
      Beep;
      { Read in the keys to clear the key buffer }
      repeat
        read(kbd,Key);
      until (not keypressed);
      DrawText(1,' ',Yes);
    end;

  procedure Next_Char;                                               {.CP15}
    { PROCEDURE TO READ THE NEXT CHARACTER FROM THE INPUT FILE }
    begin
      {Entering('Next_Char');}
      EndOfCommandLine:= (CommandPosition=CommandLength);
      if EndOfCommandLine
        then Charac:= ' '
        else
          begin
            CommandPosition:= CommandPosition + 1;
            Charac:= copy(CurrentCommand, CommandPosition, 1);
            Charac:= UpCase(Charac);
          end;
      {Exiting('Next_Char');}
    end;

  procedure Clear_Types;                                             {.CP6}
  { PROCEDURE TO CLEAR VARIABLE TYPES }
    begin
      Keyword:=false;
      Move:=false;
    end;
```

```
procedure Set_Type;                                              {.CP25}
{ PROCEDURE TO SET THE TOKEN TYPE FOR INTERNAL USE }
  begin
    Entering('Set_Type');
    Clear_Types;
    if (pos(Token, 'ARM') = 1) or
       (pos(Token, 'WRIST') = 1) or
       (pos(Token, 'DRIVE') = 1) or
       (pos(Token, 'GRIPPER') = 1) or
       (pos(Token, 'HEAD') = 1) or
       (pos(Token, 'TURN') = 1)
      then
        begin
          Move:=true;
          Keyword:=true;
        end
      else
        if (pos(Token, 'INITIALIZE') = 1) or
           (pos(Token, 'GRAPHICS') = 1) or
           (pos(Token, 'SPEAK') = 1) or
           (pos(Token, 'MEMORY') = 1) or
           (pos(Token, 'END') = 1)
          then Keyword:=true;
    Exiting('Set_Types');
  end;

procedure Put_Token;                                             {.CP6}
{ PROCEDURE TO DISPLAY TOKENS }
  begin
    if SaveTokens then writeln(Tokfile,Token);
    if Debug_Flow then writeln(lst,'Token=',Token);
  end;
```

```
procedure Next_Token;                                                {.CP46}
{ PROCEDURE TO CONSTRUCT THE NEXT TOKEN FROM THE INPUT FILE }
  type
    CharSet = set of char;
  var
    SearchSet: CharSet;
  begin
    Entering('Next_Token');
    if not EndOfCommandLine then
      begin
        Clear_Types;
        Semi_Token:='';
        Token:='';
        while (Token = '') do
          if (Charac = ' ')
            then
              begin
                if not EndOfCommandLine then Next_Char
              end
            else
              begin
                case Charac of
                  'A'..'Z'    :  SearchSet:= ['.'..':','A'..'Z'];
                  '-','0'..'9':  SearchSet:= ['.','0'..'9'];
                  '$'         :  SearchSet:= ['0'..'9','A'..'F'];
                  '.'         :  SearchSet:= ['0'..'9'];
                  end;
                Token:= Charac;
                Next_Char;
                while (Charac in SearchSet) do
                  begin
                    Token:= concat(Token,Charac);
                    Next_Char
                  end;
                Put_Token;
                if (Charac = ';') then
                  begin
                    Semi_Token:='';
                    Semi_Token:=concat(Semi_Token, Charac);
                    Next_Char
                  end;
              end;
      end;
    Set_Type;
    Exiting('Next_Token');
  end;
```

```
procedure Amount_Validate;                                    {.PA}
{
PURPOSE: Validates the values of move commands versus the limits of
the robot for each part of the robot.  This routine handles both
relative and absolute commands.  If no errors occur (i.e., if the
command is good), the position variables for the appropriate part
are updated and the proper Recalculate flags are set for only those
parts whose data points must be retransformed.
}
  var
    RP: RobotParts;
    Theta: real;
  begin
    Next_Token;
    Token_Val:= 0;
    val(Token, Token_Val, Result);

    with Parts[RobPrt] do

      if not Relative
        then                    {ABSOLUTE MODE}
          if (Result <> 0) or
             (Token_Val < Absolutes[1]) or
             (Token_Val > Absolutes[2])
            then Error(12)
            else
              begin
                Position:= Token_Val;
                RecalculateFlag:= true;
                GoodCommand:= true
              end

        else                    {RELATIVE MODE}
          if (RobPrt = Body)
            then
              if (Parts[Steering].Position <> 0.00)
                then Error(26)
                else
                  begin
                    GoodCommand:= true;
                    DrawShoulderViewFlag:= false;
                    BodyIncrement:= 12.0 * Token_Val;
                    Theta:= -Parts[Body].Position;
                    Theta:= DegToRad(Theta);
                    RobotPos[1]:= RobotPos[1] +
                                  (BodyIncrement * cos(Theta));
                    RobotPos[2]:= RobotPos[2] +
                                  (BodyIncrement * sin(Theta));
                    NewRobotPos:= true;
                  end
```

```
            else                                                    {.CP24}
              if (Result <> 0) or
                 ((Token_Val + Position) < Absolutes[1]) or
                 ((Token_Val + Position) > Absolutes[2])
                then Error(13)
                else
                  begin
                    GoodCommand:= true;
                    BodyIncrement:= 0;
                    RecalculateFlag:= true;
                    Position:= Position + Token_Val;
                  end;

    if GoodCommand then
      begin
        if RobPrt<>Steering then
          for RP:= Succ(RobPrt) to Gripm do
            with Parts[RP] do
              RecalculateFlag:= true;
        if RobPrt=Gripm
          then Parts[Gripp].Position:= Parts[Gripm].Position
          else Parts[Gripm].Position:= Parts[Gripp].Position;
      end;
  end;

procedure Motor_Select;                                             {.CP20}
{ PROCEDURE TO ASSIGN A Robot Part TO THE INPUT COMMAND }
  const
    MotorStr: array[RobotParts] of string[10] =
              ('TURN',  'DRIVE',  'HEAD',    'SHOULDER', 'EXTEND',
               'PIVOT', 'ROTATE', 'GRIPPER', 'GRIPPER');
  var
    PartFound: boolean;
  begin
    RobPrt:= Steering;
    PartFound:= false;
    repeat
      if (pos(Com, MotorStr[RobPrt]) = 1)
        then PartFound:= true
        else RobPrt:= succ(RobPrt);
    until (PartFound) or (RobPrt=Gripm);
    if not PartFound
      then Error(11)
      else Amount_Validate;
  end;
```

```
procedure Get_Modes;                                              {.CP36}
{ PROCEDURE TO OBTAIN THE MODE PARAMETERS FROM THE INPUT FILE }
  begin
    Next_Token;
    if (pos(Token, 'CONTINUE') = 1) then
      begin
        Error(31);
        Continue:=true;
      end
    else if (pos(Token, 'WAIT') = 1) then
      Continue:= false
    else Error(8);

    if (pos(Com, 'DRIVE') = 1) then
      Relative:=true
    else
      begin
        Next_Token;
        if (pos(Token, 'RELATIVE') = 1) then
          Relative:= true
        else if (pos(Token, 'ABSOLUTE') = 1) then
          Relative:=false
        else Error(9);
      end;

    Next_Token;

    if (pos(Token, 'SLOW') = 1)
      then speed:=slow
    else if (pos(Token, 'MEDIUM') = 1)
      then speed:=med
    else if (pos(Token, 'FAST') = 1)
      then speed:=fast
    else Error(10);
    Motor_Select;
  end;
```

```
procedure Move_Hero;                                              {.CP32}
{ PROCEDURE TO DEFINE THE COMMAND IF THE TOKEN IS TYPE MOVE }
  begin
    TempCom:=Token;
    if (pos(TempCom, 'ARM') = 1) then
      begin
        Next_Token;
        if ((pos(Token, 'EXTEND')=1) or (pos(Token, 'SHOULDER')=1))
          then
            begin
              Com:=Token;
              Get_Modes;
            end
          else error(7);
      end
    else if (pos(TempCom, 'WRIST') = 1) then
      begin
        Next_Token;
        if ((pos(Token, 'PIVOT') = 1) or (pos(Token, 'ROTATE')=1))
          then
            begin
              Com:=Token;
              Get_Modes
            end
          else Error(6);
      end
    else
      begin
        Com:=TempCom;
        Get_Modes
      end
  end;
```

```
procedure Graphics_Hero;                                        {.PA}
{ THE HERO GRAPHICS PROCEDURE READS GRAPHICS PARAMETERS }

  procedure GraphicsBackground;
    var
      TempStr: AnyString;
      BaseX, BaseY: real;

    function BaseFound: boolean;
    {
    PURPOSE: This procedure stores in the variable BaseX and BaseY
    the X and Y values of the Base point for the Background drawing.
    This value is signified by the word BASE on the previous line of
    drawing file.
    }
      var
        Bf: boolean;
        Comma, NumOK: integer;
        TempStr2: AnyString;
      begin
        Entering('BaseFound');
        Bf:= false;
        repeat
          readln(BgFile, TempStr);
          TempStr:= Upc(TempStr);
          if pos('BASE',TempStr)<>0 then
            begin
              readln(BgFile, TempStr);
              Comma:= pos(',',TempStr);
              if Comma<>0 then
                begin
                  TempStr2:= Copy(TempStr, 1, (Comma - 1));
                  Val(TempStr2, BaseX, NumOK);
                  if NumOK = 0 then
                    begin
                      TempStr2:= Copy(TempStr, (Comma + 1),
                                      Length(TempStr) - Comma);
                      Val(TempStr2, BaseY, NumOK);
                      if NumOK = 0 then
                        begin
                          Bf:= true;
                          BaseX:= 12 * BaseX;
                          BaseY:= 12 * BaseY;
                        end;
                    end;
                end;
            end
        until eof(BgFile) or Bf;
        Exiting('BaseFound');
        BaseFound:= Bf
      end;
```

```
function LinesFound: boolean;                                    {.CP12}
{
PURPOSE:  This procedure finds all the lines from the Background
file and stores the line X and Y coordinates into the Heap.  The
line data is signified by the word LINE stored in the previous
record of the drawing file.
}
  var
    Found, LinesError: boolean;
    Comma, I, NumOK, SpecCount: integer;
    TempReal: real;
    Number: AnyString;
  begin                                                          {.CP14}
    Entering('LinesFound');
    Found:= false;
    LinesError:= false;
    NumBgLines:= 0;
    repeat
      readln(BgFile, TempStr);
      TempStr:= Upc(TempStr);
      if pos('LINE',TempStr)<>0 then
        begin
          SpecCount:=0;
          readln(BgFile, TempStr);
          Comma:= pos(',',TempStr);
          NumOK:= 0;
          while (Comma<>0) and (NumOK=0) and (SpecCount<3) do {.CP13}
            begin
              SpecCount:= SpecCount+1;
              Number:= Copy(TempStr, 1, (Comma - 1));
              Delete(TempStr, 1, Comma);
              Val(Number, TempReal, NumOK);
              if (NumOK=0) then
                with TempBgLine do
                  if (odd(SpecCount))
                    then BgXY[SpecCount]:= 12 * TempReal - BaseX
                    else BgXY[SpecCount]:= -(12 * TempReal - BaseY);
              Comma:= pos(',',TempStr);
            end;
          if (NumOK = 0) and (SpecCount=3) then                  {.CP9}
            begin
              Val(TempStr, TempReal, NumOK);
              if NumOK = 0
                then
                  begin
                    Found:= true;
                    TempBgLine.BgXY[4]:= -(12 * TempReal - BaseY);
                    NumBgLines:= NumBgLines + 1;
```

```
            if NumBgLines = 1 then                          {.CP19}
              begin
                BgLine[NumBgLines]:= TempBgLine;
                with TempBgLine do
                  begin
                    for I:=1 to 2 do
                      begin
                        BgMaxMin[I]:= BgXY[I];
                        BgMaxMin[I+2]:= BgXY[I];
                      end;
                    for I:=3 to 4 do
                      begin
                        if BgXY[I]<BgMaxMin[I-2]
                          then BgMaxMin[I-2]:= BgXY[I];
                        if BgXY[I]>BgMaxMin[I]
                          then BgMaxMin[I]:= BgXY[I];
                      end
                  end
              end
            else if NumBgLines>MaxNumBgLines then           {.CP6}
              begin
                Error(36);
                while (not eof(BgFile)) do
                  readln(BgFile);
              end
            else                                            {.CP23}
              begin
                BgLine[NumBgLines]:= TempBgLine;
                with TempBgLine do
                  begin
                    for I:=1 to 2 do
                      begin
                        if BgXY[I]<BgMaxMin[I]
                          then BgMaxMin[I-2]:= BgXY[I];
                        if BgXY[I]>BgMaxMin[I+2]
                          then BgMaxMin[I]:= BgXY[I];
                      end;
                    for I:=3 to 4 do
                      begin
                        if BgXY[I]<BgMaxMin[I-2]
                          then BgMaxMin[I-2]:= BgXY[I];
                        if BgXY[I]>BgMaxMin[I]
                          then BgMaxMin[I]:= BgXY[I];
                      end
                  end
              end;
          end;
      end;
```

```
        if NumOK<>0 then                                    {.CP11}
          begin
            LinesError:= true;
            Error(35);
          end;
      end
    until eof(BgFile) or LinesError;

    Exiting('LinesFound');
    LinesFound:= (Found) and (not LinesError);
  end;

begin                                                       {.CP13}
  GoodCommand:= false;
  Next_Token;
  if Semi_Token<>';' then
    Error(4)
  else if (Token = 'OFF') then
    begin
      NumBgLines:= 0;
      AddBackgroundFlag:= false;
      GoodCommand:= true;
      DrawShoulderViewFlag:= false;
      DrawText(1,'GRAPHICS BACKGROUND ELIMINATED.',Yes);
    end
  else if not OpenText(BgFile, Token, true) then            {.CP11}
    begin
      FileName:= Token;
      Error(33);
    end
  else if not BaseFound then
    begin
      FileName:= Token;
      Error(34);
      close(BgFile);
    end
  else                                                      {.CP10}
    begin
      FileName:= Token;
      if LinesFound
        then
          begin
            AddBackgroundFlag:= true;
            GoodCommand:= true;
            DrawShoulderViewFlag:= false;
          end
```

```
        else                                                    {.CP9}
          begin
            NumBgLines:= 0;
            GoodCommand:= false;
          end;
      close(BgFile);
    end;
end;

procedure GraphicsLimits;                                       {.CP35}
  var
    GLimits: array[1..4] of real;
    BadLimit: boolean;
    I: integer;
  begin
    Entering('GraphicsLimits');
    I:= 0;
    BadLimit:= false;
    while (I<4) and (Semi_Token<>';') do
      begin
        Next_Token;
        I:= succ(I);
        Token_Val:= 0;
        val(Token,Token_Val,Result);
        if (Result <> 0)
          then BadLimit:= true
          else GLimits[I]:= Token_Val;
      end;
    if (BadLimit)
      then Error(28)
    else if (I<>4)
      then Error(30)
    else if (Semi_Token <> ';')
      then Error(4)
    else
      begin
        for I:=1 to 4 do
          GLimits[I]:= 12.0 * GLimits[I];
        DefineWorld(1,GLimits[1],GLimits[2],GLimits[3],GLimits[4]);
        DrawShoulderViewFlag:= false;
        GoodCommand:= true;
      end;
    Exiting('GraphicsLimits');
  end;
```

```
procedure GraphicsReverseArm;                                    {.CP35}
{ This procedure modifies a flag which indicates the direction of
the robots arm and pivoting wrist.  ArmLeft is true if the direction
of movement of the robot's arm is to the left; ArmLeft is false if the
direction of movement of the robot's arm is to the right.
This allows flexibility for the direction of movement, since the arm
can be put together to allow for motion in either direction.  Note
that if the arm moves to the left, the direction of the wrist is
initially to the left. }
  begin
    Entering('GraphicsReverseArm');
    if Semi_Token<>';'
      then
        begin
          Error(4);
          GoodCommand:= false
        end
      else
        begin
          ArmLeft:= not ArmLeft;
          DrawText(1,'THE ARM MOVES TO THE ROBOT'+chr(39)+'S ', No);
          if ArmLeft
            then DrawText(3, 'LEFT', No)
            else DrawText(3, 'RIGHT', No);
          DrawText(1, '.', Yes);
          Beep;
          GoodCommand:= true;
          DrawShoulderViewFlag:= true;
          for RobPrt:=Arm to Gripm do
            with Parts[RobPrt] do
              RecalculateFlag:= true;
        end;
    Exiting('GraphicsReverseArm');
  end;
```

```
procedure GraphicsStartingPoint;                                    {.CP46}
  var
    NewPosition: Coordinate;
    BadValue: boolean;
    I: integer;
  begin
    Entering('GraphicsStartingPoint');
    I:= 0;
    BadValue:= false;
    while (I<3) and (Semi_Token<>';') do
      begin
        Next_Token;
        I:= succ(I);
        Token_Val:= 0;
        val(Token,Token_Val,Result);
        if (Result <> 0)
          then BadValue:= true
          else NewPosition[I]:= Token_Val;
      end;
    {
    if Debug_Flow
      then writeln(lst,'BadValue=',BadValue,' Last TokenVal=',
                       Token_Val:7:2,' Semi_Token=',Semi_Token);
    }
    if (BadValue)
      then Error(29)
    else if (I<>3)
      then Error(30)
    else if (Semi_Token<>';')
      then Error(4)
    else
      begin
        RobotPos[1]:= NewPosition[1]*12.0;
        RobotPos[2]:= NewPosition[2]*12.0;
        NewRobotPos:= true;
        DrawShoulderViewFlag:= false;
        GoodCommand:= true;
        if (Parts[Body].Position<>NewPosition[3]) then
          begin
            Parts[Body].Position:= NewPosition[3];
            for RotPrt:=Steering to Gripm do
              with Parts[RotPrt] do
                RecalculateFlag:= true;
          end;
      end;
    Exiting('GraphicsStartingPoint');
  end;
```

```
  begin                                                          {.CP20}
    Entering('Graphics_Hero');
    Next_Token;
    if (pos(Token, 'BACKGROUND') = 1)
      then GraphicsBackground
    else if (pos(Token, 'LIMITS') = 1)
      then GraphicsLimits
    else if (pos(Token, 'REVERSEARM') = 1)
      then GraphicsReverseArm
    else if (pos(Token,'STARTINGPOINT') = 1)
      then GraphicsStartingPoint
    else
      begin
        Error(5);
        TokenCount:= 0;
        while (Semi_Token<>';') do
          Next_Token;
      end;
    Exiting('Graphics_Hero')
  end;

procedure End_Hero;                                              {.CP9}
{ PROCEDURE TO VERIFY END STATEMENT }
  begin
    Entering('End_Hero');
    if (Semi_Token <> ';') then Error(4);
    Done:= true;
    GoodCommand:= true;
    Exiting('End_Hero');
  end;

procedure Memory_Hero;                                           {.CP13}
{ PROCEDURE TO OUTPUT HEX MEMORY LOCATION TO OBJECT FILE }
  var
    MemStr: FontString;
  begin
    Entering('Memory_Hero');
    Next_Token;
    if (Semi_Token <> ';') then Error(4);
    MemStr:='ROBOT ADDRESS SET TO LOCATION ' + Token;
    DrawText(1,MemStr,Yes);
    PressAnyKey;
    Exiting('Memory_Hero');
  end;
```

```
procedure Speak_Hero;                                                {.CP27}
{ PROCEDURE TO ASSIGN THE LOCATION OF PRE-WRITTEN SPEECHES }
  var
    Speech_num: integer;
    SpeechStr: FontString;
  begin
    Entering('Speak_Hero');
    Next_Token;
    if (pos(Token, 'CONTINUE') = 1)
      then Continue:=true
    else
      if (pos(Token, 'WAIT') = 1)
        then Continue:= false
        else Error(2);
    Next_Token;
    val(Token,Speech_num,Result);
    if (Result <> 0) or (Speech_num < 1) or (Speech_num > 19)
      then error(3)
      else
        begin
          SpeechStr:='SPEECH COMMAND #'+Token+'; NO MOVEMENT EXECUTED.';
          DrawText(1, SpeechStr, Yes);
          PressAnyKey
        end;
    GoodCommand:= false;
    Exiting('Speak_Hero');
  end;

procedure Initialize_Hero;                                           {.CP19}
{ PROCEDURE TO MOVE ALL HERO MOTORS TO INITIAL POSITIONS }
  begin
    Entering('Initialize_Hero');
    if (Semi_Token <> ';') then Error(4);
    for RobPrt:=Steering to Gripm do
      if RobPrt<>Body then
        with Parts[RobPrt] do
          begin
            RecalculateFlag:= true;
            case RobPrt of
              Head, Wrrot: Position:= 175.0;
              Wrpvt      : Position:= 90.0;
              else         Position:= 0.0;
              end;
          end;
    GoodCommand:= true;
    Exiting('Initialize_Hero');
  end;
```

```
  procedure Controller;                                          {.CP16}
  { PROCEDURE TO CONTROL FIRST LEVEL BREAKDOWN OF COMMANDS}
    begin
      Entering('Controller');
      Set_Type;
      if not Keyword then Error(1);
      if Move then Move_hero
        else if (pos(Token, 'SPEAK') = 1) then Speak_Hero
        else if (pos(Token, 'GRAPHICS') = 1) then Graphics_Hero
        else if (pos(Token, 'INITIALIZE') = 1) then Initialize_Hero
        else if (pos(Token, 'MEMORY') = 1) then Memory_Hero
        else if (pos(Token, 'END') = 1) then End_Hero
        else if eof(source) then Error(16)
        else GoodCommand:= false;
      Exiting('Controller')
    end;

  begin                                                          {.CP8}
    Entering('Parser');
    EndOfCommandLine:= false;
    Next_Char;
    Next_Token;
    Controller;
    Exiting('Parser');
  end;

begin                                                            {.CP12}
  Debug_Flow:= false;
  Entering('ReadSourceCode');
  repeat
    GoodCommand:= false;
    GetSource;
    if not Done
      then Parser;
  until Done or Aborting or GoodCommand;
  Exiting('ReadSourceCode');
  Debug_Flow:= false;
end;
```

```
procedure TransformPoints;
{
PURPOSE: Builds transform matrices and transforms the X,Y,Z data points
according to flags and values set by Parser in ReadSourceCode.  Only the
data points for parts of the robots which moved are transformed.  The
resulting data points are such that, if drawn, the robot would be at the
0,0 coordinate (i.e., the translation of the robot to the current X and Y
coordinate is handled in the DrawOverheadView procedure).
}
  var
    RbtPrt: RobotParts;

  procedure CalculateTMatrix;
  {
  PURPOSE: Builds the transform matrix for each part of the robot.  Each
  transform matrix contains only the proper rotational and translational
  values for that specific part.
  }
    var
      I,J: integer;
      Theta, Ctheta, Stheta: real;
      Xoffset, Yoffset, Zoffset: real;
    begin
      Entering('CalculateTMatrix');
      with Parts[RbtPrt] do
        begin
          T:= identity;
          if RbtPrt=ExtendArm
            then T[4,3]:= -(Position + 4.25 + 5.25)
            else
              begin
                {
                Transform the Position to the appropriate rotation value.
                }
                case RbtPrt of
                  Gripp,Gripm:
                    begin
                      Theta:= arctan(((Position/2.0)-1.0625)/ 2.375) -
                              arctan( -1.0625 / 2.375);
                      if RbtPrt=Gripp
                        then Theta:= -Theta;
                    end;
                  Steering:
                    Theta:= -(Position+Parts[Body].Position);
                  Head, Wrrot:
                    Theta:= 175.0 - Position;
```

```
  Arm, WrPvt:                                                    {.CP7}
    if ArmLeft
      then Theta:= Position
      else Theta:= - Position;
  else
    Theta:= - Position;
  end;
{
Change Theta from degrees to radians so the Sine and Cosine
can be obtained.
}
if RotPrt<Gripp then Theta:= DegToRad(Theta);
Ctheta:= cos(Theta);
Stheta:= sin(Theta);
{
Calculate the T matrix for the part.
}
case RotPrt of
  Steering, Body, Head, WrRot:
    begin
      T[1,1]:= Ctheta;
      T[1,2]:= Stheta;
      T[2,1]:= -Stheta;
      T[2,2]:= Ctheta;
      if RotPrt=Head then
        begin
          T[4,1]:= -3.5;
          T[4,3]:= 14.5;
        end;
    end;
  Arm:                                                           {.CP15}
    begin
      Xoffset:= -0.8693;
      T[1,1]:= CphiArm;
      T[1,2]:= SphiArm * Stheta;
      T[1,3]:= -SphiArm * Ctheta;
      T[2,2]:= Ctheta;
      T[2,3]:= Stheta;
      T[3,1]:= SphiArm;
      T[3,2]:= -CphiArm * Stheta;
      T[3,3]:= CphiArm * Ctheta;
      T[4,1]:= Xoffset * CphiArm - 10.0;
      T[4,2]:= Xoffset * T[1,2];
      T[4,3]:= Xoffset * T[1,3] + 1.125;
    end;
```

```
            WrPvt:                                              {.CP7}
              begin
                T[2,2]:= Ctheta;
                T[2,3]:= Stheta;
                T[3,2]:= -Stheta;
                T[3,3]:= Ctheta;
              end;
            Gripp,Gripm:                                        {.CP15}
              begin
                Xoffset:= 1.0625;
                Zoffset:= -1.75;
                if RobPrt=Gripm
                  then Xoffset:= -Xoffset;
                T[1,1]:= Ctheta;
                T[1,3]:= -Stheta;
                T[3,1]:= Stheta;
                T[3,3]:= Ctheta;
                T[4,1]:= Xoffset;
                T[4,3]:= Zoffset;
              end;
            end;
        end;
      if Echo then                                              {.CP14}
        with Parts[RobPrt] do
          begin
            writeln(lst,'Parts[',ord(RobPrt),'].T = ');
            for I:= 1 to 4 do
              begin
                for J:=1 to 4 do
                  write(lst,' ',T[I,J]:10:5);
                writeln(lst);
              end;
          end;
    end;
  Exiting('CalculateTMatrix')
end;
```

```
procedure CalculateTfmMatrix;                                    {.CP39}
{
PURPOSE: Completes the calculation of the transform matrices for each
robot part and stores them in the global transform array Tfm.  The
matrices are completed by matrix multiplying them by all the other
transform matrices which affect that part (that is, since the rotation
of the head affects the position of the arm, the arm matrix is multiplied
by all matrices which affected the head).
}
  var
    I,J: integer;
  begin
    Entering('CalculateTfmMatrix');
    case RobPrt of
      Steering, Body:
        Tfm:= Parts[RobPrt].T;
      Gripp,Gripm:
        begin
          MatMult4x4(Parts[RobPrt].T, Parts[WrRot].T);
          Parts[RobPrt].T:= Tfm
        end;
      else
        begin
          MatMult4x4(Parts[RobPrt].T, Parts[pred(RobPrt)].T);
          Parts[RobPrt].T:= Tfm
        end;
      end;
    if Echo then
      begin
        writeln(lst,'Tfm =');
        for I:= 1 to 4 do
          begin
            for J:=1 to 4 do
              write(lst,' ',Tfm[I,J]:10:5);
            writeln(lst);
          end;
      end;
    Exiting('CalculateTfmMatrix')
  end;
```

```
procedure MatMult(Start, Finish: integer);                    {.CP33}
{
PURPOSE:  Transforms the points for a specific robot part by performing
a matrix multiply for the points by the global transform array Tfm.
The integers passed are pointers which indicate the starting and
finishing indices within the points array for the part.
}
  var
    I,J,K : integer ;
  begin
    Entering('MatMult');
    for I:=Start to Finish do
      begin
        for J:=1 to WidthArray-1 do
          begin
            XfmdPts[I,J]:= 0 ;
            for k:=1 to WidthArray do
              XfmdPts[I,J]:= XfmdPts[I,J] + Points[I,K] * Tfm[K,J];
          end;
        XfmdPts[I,WidthArray]:= 1.0;
      end;
    if Echo then
      begin
        for I:=Start to Finish do
          begin
            write(lst,'XfmdPts[',I,'] = ');
            for J:=1 to WidthArray do
              write(lst,XfmdPts[I,J]:10:5,' ');
            writeln(lst);
          end;
      end;
    Exiting('MatMult');
  end;

begin                                                         {.CP14}
  Entering('TransformPoints');
  DrawText(1,'TRANSFORMING COORDINATES.',Yes);
  for RobPrt:=Steering to Gripm do
    with Parts[RobPrt] do
      if RecalculateFlag then
        begin
          CalculateIMatrix;
          CalculateTfmMatrix;
          with Index[1] do
            MatMult(First, Last);
        end;
  Exiting('TransformPoints');
end;
```

{.PA}

```
procedure DrawOverheadView;
{
PURPOSE: Draws the Overhead View of the robot on the left side of the screen
in window 1.
}
  var
    RbtPrt: RobotParts;

  procedure OverheadViewTransform;
  {
  PURPOSE: Translates the data points to the current X and Y coordinate
  position of the robot.  The value of ScreenDistance is added to the Z
  coordinate value, then the X and Y coordinates are adjusted for
  perspective.
  }
    var
      OldPoint: Coordinate;
      I,J: integer;

    begin
      Entering('OverheadViewTransform');
      DrawText(1,'TRANSFORMING FOR OVERHEAD VIEW.',Yes);
      for RbtPrt:=Steering to Gripm do
        with Parts[RbtPrt] do
          if RecalculateFlag or NewRobotPos then
            for I:= Index[1].First to Index[1].Last do
              begin
                OldPoint:= XfmdPts[I];
                OldPoint[1]:= OldPoint[1] + RobotPos[1];
                OldPoint[2]:= -(OldPoint[2] + RobotPos[2]);
                OldPoint[3]:= -OldPoint[3] + RobotPos[3];
                AddPerspective(OldPoint);
                ViewPts[View].Vpts[I]:= OldPoint;
              end;
      Exiting('OverheadViewTransform');
    end;
```

```
procedure CorrectMaxMinForWorld;                                    {.CP40}
{
PURPOSE: Corrects the MaxMin values for the current world limit values.
If the world limit values are beyond the MaxMin values, those world values
are stored in MaxMin.  This means the area shown in the overhead view
won't change unless the world limits are changed or the robot moves
(or extends or rotates) outside those limits.  The actual world values are
not altered.
}
  begin
    Debug_Flow:= false;
    Entering('CorrectMaxMinForWorld');
    if Debug_Flow
      then
        begin
          writeln(lst,'World = ',X1WldGlb:10:4,', ',Y1WldGlb:10:4,', ',
                                  X2WldGlb:10:4,', ',Y2WldGlb:10:4);
          writeln(lst,'MaxMin= ',Xmin:10:4,', ',Ymin:10:4,', ',
                                  Xmax:10:4,', ',Ymax:10:4);
        end;
    if Xmin > X1WldGlb then Xmin:= X1WldGlb;
    if Ymin > Y1WldGlb then Ymin:= Y1WldGlb;
    if Xmax < X2WldGlb then Xmax:= X2WldGlb;
    if Ymax < Y2WldGlb then Ymax:= Y2WldGlb;

    if AddBackgroundFlag then
      begin
        if Xmin > BgMaxMin[1] then Xmin:= BgMaxMin[1];
        if Ymin > BgMaxMin[2] then Ymin:= BgMaxMin[2];
        if Xmax < BgMaxMin[3] then Xmax:= BgMaxMin[3];
        if Ymax < BgMaxMin[4] then Ymax:= BgMaxMin[4];
      end;
    if Debug_Flow
      then writeln(lst,'MaxMin= ',Xmin:10:4,', ',Ymin:10:4,', ',
                                  Xmax:10:4,', ',Ymax:10:4);
    Exiting('CorrectMaxMinForWorld');
    Debug_Flow:= false;
  end;
```

```
procedure AddBackground;                                        {.CP26}
{
PURPOSE: Adds a background picture to the overhead view.  This feature
has not yet been implemented.
}
  var
    ScBgPts: array[1..4] of integer;
    I, J, X, Y: integer;
  begin
    Entering('AddBackground');
    ClearScreen;
    for J:=1 to NumBgLines do
      begin
        with BgLine[J] do
          for I:=1 to 2 do
            begin
              Y:= (I shl 1);
              X:= -1 + Y;
              ScBgPts[X] := round(Xslope * BgXY[X] + Xint) ;
              ScBgPts[Y] := round(Yslope * BgXY[Y] + Yint) ;
            end ;
        Draw(ScBgPts[1], ScBgPts[2], ScBgPts[3], ScBgPts[4], 1);
      end;
    Exiting('AddBackground')
  end;

begin                                                           {.CP22}
  Entering('DrawOverheadView');
  View:= Overhead;
  OverheadViewTransform;
  FindHiddenLines ;
  SortPlanes;
  MaxMin ;
  TextXLcl:= TextXGlb;
  TextYLcl:= TextYGlb;
  SelectWorld(1);
  SelectWindow(1);
  CorrectMaxMinForWorld;
  Scale ;
  if AddBackgroundFlag
    then AddBackground;
  PlotPlanes;
  Beep;
  SelectWindow(3);
  TextXGlb:= TextXLcl;
  TextYGlb:= TextYLcl;
  Exiting('DrawOverheadView')
end;
```

```
procedure DrawShoulderView;                                        {.PA}
{
PURPOSE: Transforms the data and draws a view of the robot looking at the
shoulder (facing the arm).
}
  var
    RbtPrt: RobotParts;

  procedure ShoulderViewTransform;
  {
  PURPOSE: Transforms the robot data coordinates so a view from the shoulder
  is obtained.
  }
    var
      OldPoint, NewPoint: Coordinate;
      I,J: integer;
      Theta, Ctheta, Stheta: real;
    begin
      Entering('ShoulderViewTransform');
      DrawText(1,'TRANSFORMING FOR SHOULDER VIEW.',Yes);
      {
      Calculate the angle needed to view from the line shoulder.  Convert
      that angle to radians and take the sine and cosine of it.
      }
      Theta:= (175.0 - Parts[Head].Position) - Parts[Body].Position;
      Theta:= DegToRad(Theta);
      Ctheta:= cos(Theta);
      Stheta:= sin(Theta);
      {
      Transform all the points.  The rotations involved are moving the
      robot -7 along the Z axis, pitching it 90 around the x axis, yawing
      it -90 around the y axis, yawing it again by Theta, and adding the
      ScreenDistance to the z values.
      }
      NewPoint[4]:= 1.0;
      for RbtPrt:=Steering to Gripm do
        with Parts[RbtPrt] do
          if (RecalculateFlag) or (Parts[Head].RecalculateFlag) then
            for I:= Index[1].First to Index[1].Last do
              begin
                OldPoint:= XfmdPts[I];
                NewPoint[1]:= OldPoint[1] * Stheta - (OldPoint[2] * Ctheta);
                NewPoint[2]:= -OldPoint[3] + 7.0;
                NewPoint[3]:= (OldPoint[1] * Ctheta) +
                              (OldPoint[2] * Stheta) + ScreenDistance;
                AddPerspective(NewPoint);
                ViewPts[View].Vpts[I]:= NewPoint;
              end;
      Exiting('ShoulderViewTransform');
    end;
```

```
  begin                                                          {.CP23}
    Entering('DrawShoulderView');
    if DrawShoulderViewFlag
      then
        begin
          View:= Shoulder;
          ShoulderViewTransform;
          FindHiddenLines;
          SortPlanes;
          MaxMin ;
          TextXLcl:= TextXGlb;
          TextYLcl:= TextYGlb;
          SelectWindow(2);
          Scale ;
          PlotPlanes;
          SelectWindow(3);
          TextXGlb:= TextXLcl;
          TextYGlb:= TextYLcl;
        end
      else DrawText(1,'NO CHANGE IN SHOULDER VIEW.',Yes);
    Beep;
    Exiting('DrawShoulderView')
  end;

begin                                                            {.CP16}
  Entering('ReadAndDraw');
  RandDinit;
  ScreenInit;
  while not (Aborting or Done) do
    begin
      UpdateStatusLine;
      TransformPoints;
      DrawOverheadView;
      DrawShoulderView;
      ReadSourceCode;
    end;
  Exiting('ReadAndDraw');
end;

{-------------------------------------------------------------------------------}
```

{.PA}

```
procedure WrapUp;
{
PURPOSE: Prepares for ending the program.  The Source and Tokens files are
closed and the screen is restored to text.  If the program is not ending in
an abort condition, the screen is cleared.
}
  begin
    Entering('WrapUp');

    { Close the source code file. }
    if SourceOpen then close(Source);

    { Close the tokens file if necessary. }
    if (SaveTokens and TokenOpen) then close(TokFile);

    { Close the log file if necessary. }
    if LogCommands then
      begin
        writeln(LogFile);
        writeln(LogFile,ErrorCount:5,' Errors Encountered.');
        close(LogFile)
      end;

    {
    Return the screen to its normal mode.  If Aborting, don't switch screen
    modes since it clears the screen.
    }
    if not Aborting
      then TextMode(C80) ;
    TextColor(LightGray) ;
    CrtExit ;
    LowVideo ;
    Exiting('WrapUp');
  end;

{-------------------------------------------------------------------------}
```

```
                                                                    {.PA}
{-------------------------------------------------------------------------
 ************  MAIN  PROGRAM  ************
 -------------------------------------------------------------------------}

begin
  {if Debug_Flow then writeln(lst,' entering program RIGS.');}
  Initialize;
  if not Aborting then ReadAndDraw;
  WrapUp;
  {if Debug_Flow then writeln(lst,' exiting program RIGS.');}
end.
```

REFERENCES

Angell, Ian O. A Practical Introduction to Computer Graphics. New York, New York: John Wiley and Sons, Incorporated, 1981.

Artwick, Bruce A. Applied Concepts in Microcomputer Graphics. Englewood Cliffs, New Jersey: Prentice-Hall, Incorporated, 1984.

Bonner, Susan and Shin, Kang. "A Comparative Study of Robot Languages." Computer, Volume 12 (December 1982): 82-96.

Dillmann, Ruediger. "A Graphical Emulation System for Robot Design and Program Testing." Conference Proceedings, 13th International Symposium on Industrial Robots and Robots 7 Volume I, pp. 7.1-7.15. Dearborn, Michigan: Society of Manufacturing Engineers, 1983.

Foley, James and Van Dam, Andries. Fundamentals of Interactive Computer Graphics. Reading, Massachusetts: Addison-Wesley Publishing Company, 1982.

Gilber, Philip. Software Design and Development. Chicago, Illinois: Science Research Associates Incorporated, 1983.

Gruver, William A.; Soroka, Barry I.; Craig, John J.; and Turner, Timothy L., "Evaluation of Commerically Available Robot Programming Languages." 13th International Symposium on Industrial Robots and Robots 7 Volume II, pp. 12.58-12.67. Dearborn, Michigan: Society of Manufacturing Engineers, 1983.

Hartley, John. Robots at Work. Bedford, England: IFS (Publications) Ltd., 1983.

Hero Robot Model ET-18 Technical Manual. Benton Harbor, Michigan: Heath Company, 1983.

Hero Robot Model ET-18 User's Manual. Benton Harbor, Michigan: Heath Company, 1982.

Hussain, K. M. and Hussain, Donna. Information Processing Systems for Management. Homewood, Illinois: Richard D. Irwin, Inc., 1981.

Lee, C. S. G.; and Ziegler, M. "A Geometric Approach in Solving the Inverse Kinematics of PUMA Robots." Conference Proceedings, 13th International Symposium on Industrial Robots and Robots 7 Volume I, pp. 16.1-16.19. Dearborn, Michigan: Society of Manufacturing Engineers, 1983.

Larson, Thomas and Coppola, Anthony. "Flexible Language and Control System Eases Robot Programming." Electronics (June 1984): 156-159.

Salmon, Mario. "SIGLA--The Olivetti Sigma Robot Programming Language." 8th International Symposium on Industrial Robots and 4th International Conference on Industrial Robot Technology, pp. 358-363. Bedford, England: IFS (Publications) Ltd., 1978.

Washburn, Donald A. "A 'User-Friendly' Robot Operator Training Aid." Research Report, University of Central Florida, 1984.